(2)

CRITICAL PROBLEMS IN VERY LARGE SCALE COMPUTER SYSTEMS

Semiannual Technical Report for the Period

October 1, 1989 to March 31, 1990

Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Anant Agarwal	(617) 253-1448
William J. Dally	(617) 253-6043
Srinivas Devadas	(617) 253-0454
Thomas F. Knight, Jr.	(617) 253-7807
F. Thomson Leighton	(617) 253-3662
Charles E. Leiserson	(617) 253-5833
Jacob K. White	(617) 253-2543
John L. Wyatt, Jr.	(617) 253-6718

[1]This research was sponsored by Defense Advanced Research Projects Agency (DoD), through the Office of Naval Research under Contract No. N00014-87-K-0825.

Contents

1 Research Overview

The research vehicle for this contract is the largest possible computer that can be conceived for the mid to late 1990's. We call this machine an "American Resource Computer" or "ARC." We imagine this machine to occupy several floors of a building. The nation could probably only afford one or two ARC's. The machine will be used to solve large-scale scientific problems having both military and civilian applications.

This investigation addresses the hardware technology, software techniques, algorithms, communications, processing elements, and applications. The study is determining the plausibility (not feasibility) of the machine. The technical challenges of such a machine serve as our guiding stimulus for the research carried out and reported here.

The chip technology that will be available for an ARC is consistent with the following parameters, assuming a CMOS process with $\lambda = 0.125$ microns.

Size:	10m × 10m × 10m
FLOPS:	10^{15}
Bits:	10^{15}
Cost:	\$1–2 billion
Processors:	4 billion
Number of chips:	10 million
Clock:	200 MHz
Power:	100 MW (10W/chip)
Bisection bandwidth:	10^{16} bits/sec
Total node bandwidth:	10^{19} bits/sec
Component reliability:	1 hour MTBF
System reliability:	???

Research is required to overcome the numerous hurdles to making an ARC feasible. Of the issues to be faced, the most problematic is system reliability. A mean time to failure of 10^5 hours is plausible, but significant research must be done to achieve this goal economically.

Progress in the various research areas are highlighted in the forthcoming sections.

2 Circuits

A theory for the minimization of functions with multiple-valued outputs was developed [17]. Based on this theory, algorithms for optimum output encoding and state assignment have been devised. While encoding problems are typically NP-hard, the empirical behavior of these exact algorithms on typical examples indicates feasibility for medium-sized examples. Further, we believe that the exact algorithms can be used to develop more scientific methods for heuristic state assignment.

A theoretical framework for optimum finite state machine (FSM) decomposition has been laid using the notion of generalized prime implicants [7]. Under this framework, a uniform search strategy to find optimum two-way or multi-way, parallel, cascade or general decompositions under arbitrary topologies is possible. Heuristic strategies based on the exact algorithms have been developed that are computationally efficient and produce high-quality solutions. Finally, we have used factoring algorithms and the above strategies to optimize FSMs for performance, by decomposing a machine into smaller interacting machines that have smaller critical path delays. Preliminary results indicate that 20% performance improvements with 15-20% area increases are possible [28].

Work in the area of synthesis for testability of combinational circuits has targeted the synthesis of circuits with the highest possible levels of testability. Necessary and sufficient conditions for path and gate-delay-fault testability in arbitrary, multilevel networks have been determined [18]. Theoretical results relating path-delay-fault and multifault testability in arbitrary networks have been shown. Based on an understanding of the necessary conditions for delay-fault testability, synthesis procedures that have minimal area penalties that satisfy the sufficiency conditions for robust path-delay-fault and multifault testability have been developed [19]. In particular, it has been shown that a primarily used multilevel logic optimization technique, namely, algebraic factorization, retains path-delay-fault testability. Robust testing can be restrictive in some cases, therefore synthesis procedures for producing networks that are

validatable nonrobust testable have been developed [21]. Finally, preliminary results on the multifault testability of finite state machines have been obtained [6]. The work described above is summarized in [20].

A breakthrough in the area of sequential test generation has been made which allows test generation for non-scan sequential circuits with more than 1000 memory elements [23]. This algorithm exploits register-transfer-level information effectively to generate justification and differentiation sequences for stuck-at faults in the logic-level implementation of a sequential circuit. Micro-processors and Application-Specific Integrated Circuits (ASICs) can be tested without the need for Scan Design constraints using this algorithm. This algorithm achieves a 100X speed-up over previous test generation techniques on large examples.

An algorithm for formally verifying the equivalence of two sequential machines described at the logic level has been developed [24]. This strategy is based on a state differentiation algorithm – the equivalence of two FSMs can be posed as "Is there a differentiating sequence for the reset states of the two machines?". The algorithm can be generalized to verify hierarchical representations of sequential circuits. Pipeline latches may result in much larger State Transition Graphs that can easily be incorporated. This approach is viable for circuits of larger size than previous approaches.

Alexander Ishii has been working with Thomas Knight on a self-terminating, digitally-controlled, and ECL-compatible output pad driver for high speed integrated circuits. By automatically series-terminating driven lines with their characteristic impedances, the driver realizes speed, power, and noise improvements over conventional designs. Series termination is realized by exploiting the intrinsic series resistance of the output drive transistors. A previous design used analog circuit techniques to control the gate voltage, and thus the resistance, of the drive transistors, while the current design controls series resistance by using a 7-bit digital signal to vary the total width of the drive transistors. The design has not yet been fabricated, but simulations indicate that data-transition rates in excess of 100MHz are possible.

3 Processing Elements

Anant Agarwal, Beng Hong Lim, and John Kubiatowicz have designed APRIL, a multithreaded VLSI processor with high single-thread performance. John Kubiatowicz is coordinating its initial implemented by modifying a SPARC processor through an LSI Logic collaboration. The ASIM simulator includes a simulation module for the APRIL processor. The modifications to SPARC will help support rapid context switching and efficient trap handling, and fine-grain synchronization using full-empty bits. A multithreaded processor mitigates the negative effects of long communication and synchronization delays in multiprocessors by overlapping these delays with computation from other processes.

Anant Agarwal evaluated the performance of multithreaded processors in large scale multiprocessors using an analytical model. For processor parameters derived from APRIL's SPARC-based implementation, the study showed that multithreaded processors such as APRIL can achieve over 80% efficiency with just three threads with a 10 cycle memory delay in a cube network with base average latency of 55 cycles.

Kiyoshi Kurihara developed a new method of trace-driven performance evaluation that eliminates the correctness problems of earlier trace-driven simulation methods. This scheme schedules the simulation to obey synchronization constraints in the parallel program using synchronization information stored in the trace.

The processors of a multicomputer require the ability to switch tasks rapidly to hide transmission latency without sacrificing single-thread performance. Peter Nuth and Bill Dally have been working on an architecture for a named state processor that achieves this goal by explicitly binding names to all processor registers and interleaving tasks on a microcycle basis. This mechanism combines the advantages of multi-threading and multiple register sets for implementing fast context switches and procedure calls. It also provides a general synchronization mechanism.

Over the past six months work has concentrated on studying methods of binding names to machine registers. They have discovered that using a context cache significantly improves register storage utilization over fixed register window methods and allows fast context switches between a large set of tasks. Design studies have been carried out to assess the impact of using a register context cache on area and performance.

Scott Wills and Bill Dally have been working on a parallel architectural interface for multi-model ex-

ecution. Most multicomputers are specialized to execute a single model of computation (e.g., dataflow, actors or shared memory). They have identified a set of primitive mechanisms for communication, synchronization and naming that are required for all of these models of computation. They are currently evaluating these mechanisms in terms of their implementation cost and their suitability for supporting popular models of parallel computation.

During the reporting period they have developed an interface definition. A simulator has been constructed to test this interface. Several examples of model mechanisms have been demonstrated using the interface including: shared memory (with caches), set synchronization, object name translation, and non-resident handler support. In addition, two applications (n-body and relaxation) have been implemented on the interface. Work is underway to specify a machine to support the interface. Several hardware requirements of the interface are being incorporated into this design including: low latency communication, fast context switching, low cost synchronization, ability to exploit locality, and efficient support for sequential code sequences.

4 Communications Topology and Routing Algorithms

Bill Dally and his group have been investigating methods for improving the performance and reliability of multicomputer interconnection networks. Their recent work has concentrated on: virtual channel flow-control to improve network throughput, the express cube topology to push throughput and latency to their physical limits, and deadlock-free adaptive routing methods to perform load balancing and achieve fault tolerance. They are also starting to investigate the architecture of generalized routers that are capable of supporting many different topologies, routing strategies, and flow-control methods.

Bruce Maggs, Sanjeev Arora, and Tom Leighton have been studying nonblocking networks. Nonblocking networks arise in a variety of applications involving communications. The most well known examples include telephone networks, data networks, and distributed memory architectures. Although asymptotically optimal constructions were known for nonblocking networks, it was not known how to select paths for the desired network connections efficiently on-line. This past fall, they discovered the first optimal-time algorithms for path selection in an optimal-size nonblocking network. In particular, they showed how to route any sequence of connections and disconnections among N terminals in a multi-Benes network with $O(\log N)$ bit-step delay. Viewed in the context of a telephone switching network, their network and algorithm can handle any sequence of calls among N parties with $O(\log N)$ bit step delay per call (even if many calls are made at once). Parties can hang up and call again whenever they like, and multiparty calls can be made without affecting the performance of the algorithm. Every call still gets through in $O(\log N)$ time. Viewed in the context of distributed memories for parallel machines, their algorithm allows any processor to access any idle block of memory within $O(\log N)$ bit steps at any time, no matter what other connections have been made previously or are being made simultaneously. Lastly, all of these results still hold even if large numbers of switches in the network become faulty.

Prof. Leighton also made substantial progress on the design and analysis of routing algorithms for commonly used networks such as arrays and hypercubes. For arrays, Prof. Leighton developed techniques for analyzing the average case behavior of routing algorithms in a variety of models. Some of the results are quite strong. For example, a variation of the greedy algorithm similar to that used in architectures designed by Seitz and Dally is shown to route most all the messages to their destination with only constant delay.

Bruce Maggs, Bill Aiello, Tom Leighton, and Mark Newman have been studying algorithms for bit-serial routing on a hypercube. They have developed a randomized adaptive algorithm that routes any permutation of $O(\log N)$-bit packets on an N-node hypercube in $O(\log N)$ bit-steps, with high probability. Furthermore, they have showed that any randomized oblivious algorithm requires $\Omega(\log^2 N/\log\log N)$ bit-steps for most permutations.

Shlomo Kipnis continued his investigation of priority arbitration schemes that employ m busses to arbitrate among n modules. His binomial arbitration scheme, which uses $m = \lg n + 1$ busses, enables achieving an arbitration time of $t = \frac{1}{2}\lg n$ (in units of bus-settling delay). Furthermore, his generalized binomial arbitration scheme achieves a bus-time tradeoff of the form $m = \Theta(tn^{1/t})$ between the number of arbitration busses m and the arbitration time t, for values of $1 \leq t \leq \lg n$ and $\lg n \leq m \leq n$. These

schemes can be adopted with no changes to existing hardware and protocols; they merely involve selecting a good set of priority arbitration codewords [26]. Shlomo Kipnis had applied for a patent, through the MIT Technology Licensing Office, on these new arbitration schemes.

5 Systems Software

Anant Agarwal and his students have continued studying methods of exploiting locality for scalability in large-scale multiprocessors. They have investigated the use of caches in providing efficient coherent shared memory. David Chaiken has designed the cache coherence protocol specification for limited directory and single-link chained schemes, and John Kubiatowicz is focusing on the VLSI implementation of these protocols. The protocols have been implemented in ASIM, which is a large scale multiprocessor simulator.

Several parallel applications have been written, successfully compiled, and run on the ASIM simulator. The simulator has been heavily instrumented and yields a wide range of useful statistics including parallelism profiles, communication locality histograms, cache and network statistics, processor utilization, process length distributions, run lengths between synchronizations, etc. These execution profiles provide feedback to the programmer and help in parallel program optimization.

ASIM now includes many new features such as a floating-point coprocessor, support for special processor mechanisms including remote process invocation, full-empty bit synchronization with support for arrays of full-empty bit data, chained directory coherence protocols, cache-only-private-data coherence protocol, software/hardware coherence protocol interactions, and optimizations for cache and directory data structures.

David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal evaluated the effectiveness of caches in multiprocessors using trace driven simulation of several large parallel applications including Speech, SIMPLE, WEATHER, and LocusRoute. The study showed that cache-based memory systems are viable for large-scale multiprocessors, and that relatively small caches (16K) are adequate to ensure good performance. They showed that limited directory and chained coherence protocols are both memory efficient and scalable. These simulations established some basic principles for implementing a cache-based multiprocessor system. For example, the results demonstrated that the best overall performance of multiprocessors will result only if software systems are optimized for caches. Several applications (e.g., WEATHER, SIMPLE) required new software-combining-tree structures for their barrier synchronizations. The Speech application required identification of read-only data words for special handling.

The above study contradicted the widely held belief that large caches (say 256K–1M bytes) are mandatory for good performance. They found that for all their parallel benchmark applications including Speech, SIMPLE, WEATHER, and LocusRoute, the performance of cache-based systems was virtually indistinguishable for cache sizes greater than 16K bytes. The reason is that individual process working sets become smaller as parallelism at finer grain is exploited, making the cache-size-dependent miss rate component a small fraction of the overall cache miss rate. The above result is important because small on-chip VLSI caches can be clocked at much higher rates and are easier to implement than large off-chip caches.

David Kranz has retargeted the Orbit optimizing compiler for the APRIL processor. Kranz has also implemented a novel method of dynamic process partitioning called *Lazy Futures*. The Lazy Futures method virtually obviates the overhead associated with task creation and deletion when tasks run on the same processor on which they were created. For example, with Lazy Futures, the sequential version of an application runs at roughly the same speed as a single processor execution of a parallel version of the same application.

Dan Nussbaum is working on a runtime system that optimizes locality of memory referencing through clever scheduling methods. Enhancing communication locality is of fundamental importance if parallel computers are to scale. Nussbaum has implemented a tree scheduler together with lazy task creation to manage parallelism and locality. The scheduler is operational on their simulator. The tree-scheduler is currently being augmented with support for multithreaded processor operation.

They have made several linguistic extensions to their Mul-T programming language to support static scheduling of processes, memory allocation, and data parallel operation. They have added a dataflow I-structure-like facility called J-structures for avoiding barrier synchronization the the overhead of Futures. These facilities have been implemented in the Orbit compiler and the ASIM simulator by Lim and Kranz.

Kirk Johnson's and Beng Hong Lim's initial applications efforts using this synchronization mechanism shows significant benefits over naive barrier synchronization. For example, the Multigrid application showed close to a factor of two improvement in speed when its Futures were replaced with the fine-grain J-structures.

Gino Maa has been working on compiler methods for enhancing locality in parallel programs. The idea is to statically partition code and allocate data structures based on minimization of non-local memory references. The particle-in-cell (PIC) program, written originally by Olaf Lubek, has been parallelized and ported to Mul-T. It is being examined closely as an instance of a massively data-parallel application suitable for fine-grained architectures. Structurally, PIC consists of steps of operations which produce successive intermediate matrices. The operations typically read from a few locations of the input matrices and compute some locations for the output matrices, which is used as the input to some subsequent operations. Fine-grain architectures synchronize on the individual elements instead of on entire matrices, thereby exposing the producer-consumer parallelism. Because of the relatively trivial amount of work involved in each of these operations (usually a sum of few products), the overhead of scheduling and synchronization becomes prohibitively high. To reduce these and communication (i.e., improving locality) costs, it will be beneficial to merge some of the operations together: by merging an upstream operation with a downstream one, the intermediate matrix element can potentially be made local, eliminating BOTH the communication and synchronization costs. By merging two peer operations which share some of their input matrix element accesses, the amount of network traffic can be reduced again. The merging increases the task granularity, thus reducing the number of tasks which needs to be scheduled at runtime and the total scheduling overhead. The desired granularity can be adjusted to match the system size (i.e., number of processors) and the actual cost structure of the runtime and hardware to insulate the programmer from details of the runtime environment. Moreover, once most of the accesses to a matrix element are from a particular task, it starts to make sense to allocate the corresponding matrix elements and tasks to the same processor node.

Andrew Chien and Bill Dally have been working on abstraction tools for programming message-passing multicomputers. They have developed a programming language, Concurrent Aggregates (CA), that contains support for building massively concurrent data abstractions. Traditional abstraction techniques only allow limited concurrency for each abstraction. This is due to the sequentializing semantics of the data abstraction tools. Their aggregates – homogeneous collections of objects – can be used to implement data abstractions with virtually unlimited concurrency. In addition, sequentialized data abstraction's implementations may be more efficient because programmers can explicitly control replication and consistency.

They have implemented a compiler and run-time system for Concurrent Aggregates. A large number of application kernels have been written using this system. Their implementation compiles programs for a message passing machine simulator. This simulator models an abstract fine-grained message passing machine with similar executions costs to the J-machine. The Concurrent Aggregates programming system has been distributed to several sites.

Their application kernels include multigrid relaxation algorithm, A* search, N-body interaction, and a digital logic simulator. They have simulated program executions consisting of tens of millions of message passing operations. Their initial simulations show that Concurrent Aggregates programs can exhibit massive concurrency and good efficiency. Programming experience with CA has shown that their novel data abstraction tools can reduce the complexity of programs by allowing abstractions to be used without causing a reduction in concurrency. They are currently performing a more extensive evaluation — programming and simulation — of Concurrent Aggregates programs.

6 Algorithms

Under the supervision of Charles Leiserson, Marios Papaefthymiou has been investigating efficient algorithms for optimization of synchronous systems. He presented a concise mathematical characterization of the minimum clock period of a synchronous system in terms of the minimum register-to-delay ratio cycle in the system's graph representation. Specifically, he showed that the minimum clock period of a system with components of arbitrary delay cannot exceed this ratio by more than the maximum delay D of the components. These results led to improved algorithms for retiming of *arbitrary* delay systems: an

$O(V^{1/2}E \lg VW \lg VD)$ algorithm for retiming with clock period that does not exceed the minimum by more than D, and an $O(VE \lg D)$ algorithm for minimum clock period retiming.

Alexander Ishii has generalized his VLSI timing analysis algorithms using the notion of a "base step" function to encapsulate assumptions about when signal values change during the operation of a circuit. He has shown how various base step functions can be used to provide sufficient conditions for a circuit to operate properly. The base step function is used to derive a "computational expansion" of the circuit from which a collection of simple linear constraints are derived. These constraints can be efficiently checked using standard graph algorithms. In addition, the algorithm can be adapted to determine the maximum frequency at which a circuit can be clocked and to produce the limiting critical path.

Ishii and Leiserson have also developed a new base step function which is less pessimistic than the ones used in previous timing verifiers, yet correctly handles timing constraints that are "cyclic" or extend across the boundaries of multiple clock phases or cycles. If a circuit is modeled as a graph $G = (V, E)$, where V consists of components—latches and functional elements—and E represents intercomponent connections, the new base step function results in an algorithm which verifies the proper timing of a circuit in worst-case $O(|V||E|)$ time and $O(|V|^2)$ space [25].

Shlomo Kipnis finished compiling a survey paper on the problem of range queries in computational geometry. Range queries are a fundamental problem in computational geometry with applications to computer graphics and database retrieval systems. The survey paper identifies three general methods for range queries in computational geometry and classifies many of the recent research results into one or more of these methods [56].

During the past six months, James K. Park has been collaborating with Alok Aggarwal, Dina Kravets, and Sandeep Sen on a number of problems relating to Monge arrays. Aggarwal and Park have been studying the use of Monge arrays in solving economic lot-size problems arising in operations research. Aggarwal, Kravets, Park, and Sen have been investigating parallel alogorithms for searching in staircase-Monge arrays and the conversion of PRAM algorithms for searching in Monge arrays to algorithms for hypercubes and related interconnection networks.

Park has also been working on his doctoral thesis, tentatively titled *The Monge Array — An Abstraction and Its Applications*. The thesis, a comprehensive study of Monge arrays and their applications, should be completed by August 1990.

Tom Cormen, Charles Leiserson, and Ron Rivest have completed the textbook *Introduction to Algorithms*. The book will be published in April 1990.

7 Applications

A database management system (DBMS) intended for the highly concurrent J-Machine system is being formulated by John Keen and Bill Dally. It will permit many transactions to simultaneously operate on a database. Prominent research issues are concurrency control, logging and recovery in a highly concurrent system. A prioritorized pre-emptive locking scheme has been proposed as a suitable concurrency control technique. A set of log processors operate in parallel to collect log fragments for updates to data and transaction commits. Recovery from these logs is performed incrementally so that normal processing can resume as quickly as possible after a crash. These techniques for designing a concurrent DBMS will offer the possibility of scaling up the system hardware and software to handle transaction processing rates much higher than those achievable with high performance expensive serial systems.

Over the past six months, Jacob White and his students have continued efforts in developing numerical algorithms for problems related to the design of an ARC, as well as those that can effectively exploit the ARC's capability. The emphasis is now shifting from unearthing entirely new approaches to extending those approaches to solve wider classes of real problems and to developing solid programs others can use. This is particularly true of their work on capacitance calculation and mixed circuit-device simulation. Some completely new avenues are still being investigated, particularly in the area of Monte-Carlo device simulation and in techniques for peak power and current estimation for large digital circuits.

Three dimensional capacitance and inductance extraction has recently become important because the dense packing of processors and memory required for high performance parallel computers require three dimensional interconnection. A fast algorithm for computing the capacitance of a complicated 3-D ge-

ometry of ideal conductors in a uniform dielectric has been developed and tested [44]. The method is an acceleration of the standard integral equation approach for multiconductor capacitance extraction. These integral equation methods are slow because they lead to dense matrix problems which are typically solved with some form of Gaussian elimination. This implies the computation grows like n^3, where n is the number of tiles needed to accurately discretize the conductor surface charges. Professor White and his students have developed a preconditioned conjugate-gradient iterative algorithm with a multipole approximation to compute the iterates. This reduces the complexity of the multiconductor capacitance calculations to grow as nm where m is the number of conductors. Their most recent efforts are in improving the implementation of the capacitance calculation algorithm. Specifically, they have developed a faster code which includes a novel adaptive 3-D multipole algorithm. Their future work in this subject will be to include dielectric interfaces and then they will turn their attention to developing a multipole algorithm to accelerate the calculation of inductances.

In the area of circuit simulation, Professor White and his students have completed the development of SIMLAB [60, 59], a fast, general purpose circuit simulation program intended to facilitate research and which has also been used to teach Professor White's course in numerical simulation. SIMLAB is also being used to investigate techniques for simulation of "vision circuits," which are fairly regular, but very large analog circuits which are very expensive to simulate using programs like SPICE. Over the past six months, vision simulation extensions have been added to SIMLAB, and the program was used to investigate the behavior of a class of nonlinear smoothing and segmentation networks. In addition, several conjugate-gradient and waveform-Newton [50] based solution algorithms have been added to SIMLAB, and their performance was tested on resistive grid and vision circuits. Finally, they are investigating what they hope is a very novel and very general extension to the algebraic multigrid algorithm suitable for both serial and parallel vision circuit simulation.

Also in circuit simulation, over the past few months Professor White and his colleagues have completed a theoretical study of exponential-Fitting numerical integration algorithms [45]. They have been able to prove several strong results that indicate the performance of recently published exponential-fitting algorithms are, in the limit of large timesteps, identical to other well-known techniques. They have also found through experiments that a simple and somewhat subtle modification stabilizes one of the more unstable — but when stable more accurate — expontential-fitting methods, and they are now trying to prove this will always be the case.

In the area of classical device simulation, Professor White is trying to finish his work on waveform relaxation for 2-D MOS device simulation, and the straight-forward extension to 3-D [11, 48, 49]. Current results indicate that the WR algorithm can be an order of magnitude faster than standard techniques for transient simulation, and the code is to be improved by investigating iterative mesh refinement, improving the matrix solution code, and adding more realistic mobility models. In addition, he and his collaborators are investigating efficient techniques for computing terminal currents in preparation for using the simulator in a mixed circuit-device simulation program. Finally, they have arranged for access to an INTEL hypercube in order to demonstrate what they hope will be WR's strongest benefit, that of being easily and effectively parallelized.

Although still useful for predicting terminal currents, the drift-diffusion model of electron transport does not accurately predict the field distribution near the drain in small geometry devices. This is of particular importance for predicting oxide breakdown due to penetration by "hot" electrons. There are two approaches for more accurately computing the electric fields in MOS devices: one is based on adding an energy equation to the drift-diffusion model, and the second is based on particle or Monte-Carlo simulations.

In the first approach, an energy balance equation is solved along with the drift-diffusion equations so that the electron temperatures are computed accurately. This combined system is numerically less tame than the standard approach, and must be solved carefully. Professor White and his students have developed and tested a 2-D simulator for MOS devices based on the drift-diffusion plus energy equations, and have uncovered a source of instability in the computation when the semiconductor mobility is made a function of the inverse of carrier temperature. Although using a very fine mesh removes this instability, and duplicates what others have done, this makes the method computationally much more expensive. They are currently trying to understand the cause of the instability and correct it, so that a coarse mesh can be used and still produce accurate results.

In the area of Monte Carlo device simulation, Professor White and his students are focussing on transient calculations with self-consistent electric fields. Specifically, they are trying to apply the recently developed implicit particle methods. To apply these implicit particle methods to semiconductors, they are decomposing the field calculation into a part due to charged particles, a part due to dopant ions, and a part due to boundaries. This allows the calculation of the electric field acting on every charged particle in the system to be performed rapidly and accurately using fast multipole algorithms. Currently, they have rewritten (with J. Phillips) a Silicon Monte Carlo code from the National Center for Computational Electronics to use ensemble Monte Carlo methods and are now including the electric field calculations.

The high transistor density now possible with CMOS integrated circuits has made peak power dissipation and peak current density important design considerations. However, peak quantities in a logic circuit are usually a function of the input vector or vector sequence applied. This makes accurate estimation of peak quantities extremely difficult, since the number of input sequences that have to be simulated in order to find the sequence that produces the peak is *exponential* in the number of inputs to the circuit. By using simplified models of power and current dissipation, peak quantities — like power or current density — can be related to maximizing gate output activity, a weighted to account for differing load capacitances or transistor sizes. Transformations can then be derived that convert a logic description of a circuit into a multiple-output Boolean function of the input vector or vector sequence, where each output of the Boolean function is associated with a logic gate output transition. It then follows that to find the input or input sequence that maximizes the quantity of interest, a *weighted max-satisfiability* problem must be solved. For the problem of estimating peak power dissapation, algorithms for constructing the Boolean function for dynamic CMOS circuits, as well as for static CMOS, which take into account dissipation due to glitching, have been derived and exact and approximate algorithms for solving the associated weighted max-satisfiability problem have been developed [22].

8 Publications

8.1 Journal and Conference Publications

[1] Anant Agarwal and Anoop Gupta. Temporal, processor, and spatial locality in multiprocessor memory references. In Stu Tewksbury, editor, *Frontiers in Computing Systems Research*. Plenum Press, 1989. Also to appear as MIT VLSI Memo.

[2] Anant Agarwal and Minor Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings of ACM Sigmetrics 1990*, May 1990. To appear.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. To appear in *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[4] Prathima Agrawal and William J. Dally. A hardware logic simulation system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 19–29, January 1990.

[5] S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a non-blocking network. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, May 1990. To appear.

[6] P. Ashar, S. Devadas, and A. R. Newton. Multiple-fault testable sequential machines. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990. To appear.

[7] P. Ashar, S. Devadas, and A. R. Newton. A unified approach to decomposition and re-decomposition of sequential machines. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[8] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache-coherence in large-scale multiprocessors. To appear in *IEEE Computers*, June 1990.

[9] Andrew A. Chien and William J. Dally. Concurrent aggregates (CA). In *1990 ACM Symposium on the Principals and Practice of Parallel Programming*, 1990. to appear.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[11] M. Crow, J. White, and M. Ilic. Stability and convergence aspects of waveform relaxation applied to power system simulation. In *Proc. Int. Symp. on Circuits and Systems*, Portland, Oregon, May 1989.

[12] William J. Dally. Fine-grain concurrent computing. In Albert R. Meyer, et al., editor, *Research Directions in Computer Science: an MIT Perspective*. MIT Press, 1990.

[13] William J. Dally. The J-Machine system. In Patrick H. Winston, with Sarah A. Shellard, editor, *Artificial Intelligence at MIT; Expanding Frontiers*, pages 536–569. MIT Press, 1990. Volume 1.

[14] William J. Dally. *Network and Processor Architecture for Message-Driven Computers*. Morgan Kaufmann Publishers, 1990.

[15] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 1990. To appear. Also to appear in *Artificial Intelligence at MIT; Expanding Frontiers*, edited by Patrick H. Winston, with Sarah A. Shellard, MIT Press, 1990, volume 1, pages 508–535.

[16] William J. Dally. Virtual-channel flow control. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, 1990. To appear.

[17] S. Devadas. Minimization of functions with multiple-valued outputs: Theory and applications. In *Proceedings of the 20th Int'l Symposium on Multiple-Valued Logic*, May 1990. To appear.

[18] S. Devadas and K. Keutzer. Necessary and sufficient conditions for robust delay-fault testability of combinational logic circuits. In *Proceedings of the Sixth MIT Conference On Advanced Research in VLSI*, April 1990. To appear.

[19] S. Devadas and K. Keutzer. Synthesis and optimization procedures for robust delay-fault testability of logic circuits. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[20] S. Devadas and K. Keutzer. Synthesis for testability: A brief survey. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990. To appear.

[21] S. Devadas and K. Keutzer. Validatable nonrobust delay-fault testable circuits via logic synthesis. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990. To appear.

[22] S. Devadas, K. Keutzer, and J. White. Estimation of power dissipation in CMOS combinational circuits. In *Proc. Custom Int. Circuits Conf.*, Boston, 1990. To appear.

[23] A. Ghosh, S. Devadas, and A. R. Newton. Sequential test generation at the register-transfer and logic levels. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[24] A. Ghosh, S. Devadas, and A. R. Newton. Verification of interacting sequential circuits. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[25] Alexander T. Ishii and Charles E. Leiserson. A timing analysis of level-clocked circuitry. In *Sixth MIT Conference on Advanced Research in VLSI*, April 1990.

[26] Shlomo Kipnis. Priority arbitration with busses. Technical Memorandum TM-408, MIT Laboratory for Computer Science, October 1989. To appear in the Sixth MIT Conference on Advanced Research in VLSI, April 2–4, 1990.

[27] D. Kravets and J. Park. Selection and sorting in totally monotone arrays. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 494–502, January 1990.

[28] K. Lam and S. Devadas. Performance-oriented synthesis of finite state machines. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990. To appear.

[29] T. Leighton. A $2d - 1$ lower bound for 2-layer knock-knee channel routing. *SIAM J. Discrete Math.* 1990. To appear.

[30] T. Leighton. Average case analysis of greedy routing algorithms on arrays. In *2nd ACM SPAA*, January 1990. To appear.

[31] T. Leighton, A. Aggarwal, and M. Hansen. Solving query-retrieval problems by compacting Voronoi diagrams. In *Proc. 22nd ACM Symp. on Theory of Computing*, May 1990. To appear.

[32] T. Leighton, A. Aggarwal, and K. Palem. Area-time optimal circuits for iterated addition in VLSI. *IEEE Trans. on Computers*, 1990. To appear.

[33] T. Leighton, S. Arora, and B. Maggs. On-line algorithms for path selection in a nonblocking network. In *Proc. 22nd JACM Symp. on Theory of Computing*, May 1990. To appear.

[34] T. Leighton, B. Berger, M. Brady, and D. Brown. Nearly optimal algorithms and bounds for multilayer channel routing. *JACM*, 1990. To appear.

[35] T. Leighton, S. Bhatt, F. Chung, J. Hong, and A. Rosenberg. Optimal simulations of butterfly networks. *JACM*, 1990. To appear.

[36] T. Leighton, E. Coffman, and L. Flatto. First-fit allocation of queues: Tight probabilistic bounds on wasted space. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, January 1990. To appear.

[37] T. Leighton and B. Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. 30th IEEE Conf. on Found. of Comp. Sci.*, pages 384–389, October 1989.

[38] T. Leighton, B. Maggs, and M. Newman. Fast algorithms for bit-serial routing on a hypercube. In *2nd ACM SPAA*, January 1990.

[39] T. Leighton, F. Makedon, and D. Pathria. Efficient reconfiguration of WSI arrays. In *Proc. 1st ACM/IEEE Int. Conf. on System Integration*, 1990. To appear.

[40] T. Leighton, F. Makedon, and I. Tollis. A $2n - 2$ step algorithm for routing on an $n \times n$ array with constant size queues. *Algorithmica*, 1990. To appear.

[41] T. Leighton, F. Makedon, and S. Tragoudas. Approximation algorithms for VLSI partition problems. In *Proc. IEEE Int. Symp. on Circuits and Systems*, 1990. To appear.

[42] T. Leighton and P. Shor. Tight bounds for minimax grid matching, with applications to the average case analysis of algorithms. *Combinatorica*, 9(2):161–187, 1989.

[43] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel tasks. To appear in the LISP and Functional Programming Conference, August 1990.

[44] K. Nabors and J. White. A fast multipole algorithm for capacitance extraction of complex 3-D geometries. In *Proceedings Custom Int. Circuits Conf.*, May 1989. Updated December, 1989.

[45] H. Neto, L. Silveira, J. White, and L. Vidigal. On exponential fitting for circuit simulation. In *Proc. Int. Symp. on Circuits and Systems*, 1990. New Orleans.

[46] Michael Noakes and William J. Dally. System design for the J-Machine. In *Sixth MIT Conference on Advanced Research in VLSI*, 1990.

[47] Dan Nussbaum, Ingmar Vuong, and Anant Agarwal. Modeling a circuit-switched multiprocessor interconnect. In *Proceedings of ACM Sigmetrics 1990*, May 1990. To appear.

[48] M. Reichelt and J. White. Techniques for switching power converter simulation. In *NASECODE Conference*, Dublin, Ireland, June 1989. Invited Paper.

[49] M. Reichelt, J. White, and J. Allen. Waveform relaxation for transient simulation of two-dimensional MOS devices. In *Proc. Int. Conf. on Computer-Aided Design*, October 1989. Santa Clara, California.

[50] R. Saleh, J. White, A.L. Sangiovanni-Vincentelli, and A. R. Newton. Accelerating relaxation algorithms for circuit simulation using waveform-newton and step-size refinement. *IEEE Transactions on Computer-Aided Design*, October 1990. To appear.

8.2 Internal Memoranda

[51] Anant Agarwal. Limits to network performance, or, moderate dimensions are better. To appear as a MIT VLSI Memo.

[52] Anant Agarwal. Performance tradeoffs in multithreaded processors. To appear as a MIT VLSI Memo.

[53] A. Aggarwal, D. Kravets, J. Park, and S. Sen. Parallel searching in generalized Monge arrays with applications. Unpublished manuscript. Submitted to *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990.

[54] A. Aggarwal and J. Park. Applications of array searching to economic lot-size problems. Technical Report. To appear as a MIT VLSI Memo, 1990.

[55] B. Aiello, T. Leighton, B. Maggs, and M. Newman. Fast algorithms for bit-serial message routing on a hypercube. Internal Memorandum.

[56] Shlomo Kipnis. Three methods for range queries in computational geometry, November 1989. Submitted to *ACM Computing Surveys*.

[57] T. Leighton, M. Hansen, and C. Leiserson. Lecture notes for 18.436/6.849 *Advanced Parallel and VLSI Computation*. Technical Report MIT-LCS-RSS-7, MIT Laboratory for Computer Science, 1989.

[58] T. Leighton, B. Maggs, A. Ranade, and S. Rao. Randomized algorithms for routing and sorting in fixed-connection networks. *J. Algorithms*, 1990. Submitted for publication.

[59] A. Lumsdaine, L. Silveira, and J. White. Simlab programmers guide. Technical report, VLSI Memo, 1990. To appear.

[60] A. Lumsdaine, L. Silveira, and J. White. Simlab users guide. Technical report, VLSI Memo, 1990. To appear.

8.3 Talks without Proceedings

1. Anant Agarwal, "Processor Architecture for Multiprocessing," *Workshop on Multithreaded Processors*, Laboratory for Computer Science, MIT, January 1990.

2. Bill Dally, "Express Cubes: Improving the Performance of k-ary n-cube Interconnection Networks" *DARPA Contractors Meeting*, Washington, D.C., November, 1989.

3. Bill Dally, "Express Cubes: Improving the Performance of k-ary n-cube Interconnection Networks, and The J-Machine" *Cornell University Colloquium*, Ithaca, NY, November 1989.

4. Bill Dally, "Parallel Processing: State-of-the-Art Review" *1989 DSD Symposium*, IBM, Poughkeepsie, New York, October 1989.

5. Bill Dally, "Processor Design for Concurrent Computers," *Canadian VLSI Conference*, Vancouver, British Columbia, October 1989.

6. Bill Dally, "The J-Machine: A Fine-Grain Concurrent Computer" *Hudson Technical Seminar Series*, Digital Equipment Corporation, Hudson, MA, December 1989.

7. S. Devadas, Panel Discussion on "System-Level Verification," *IEEE VLSI Workshop*, Tampa, February 1990.

8. S. Devadas, "Panel Discussion on "Synthesis for Testability," *Int'l Conference on Computer Design: VLSI in Computers*, Cambridge, October 1989.

9. S. Devadas, "Tutorial on "New Trends in Testing and Verification," *Int'l Conference on Computer-Aided Design*, Santa Clara, November 1989.

10. S. Devadas, "Logic Synthesis and Testing Research at MIT," *Princeton University*, Princeton, November 1989.

11. S. Devadas, "Sequential Logic Verification," *1989 Boulder Summer Logic Synthesis Workshop*, January 1990.

12. S. Devadas, "Synthesis for Combinational and Sequential Testability," *1989 Boulder Summer Logic Synthesis Workshop*, January 1990.

13. S. Devadas, "Synthesis for Testability" *DARPA Microsystems and Prototyping Meeting*, Washington D.C., November 1989.

14. Alexander Ishii and Charles Leiserson, "A Timing Analysis of Level-Clocked Circuitry," *University of California at Berkeley*, March 1990.

15. Kirk Johnson, "Exploiting Concurrency in Voyager's Lexical Access Subsystem." *Spoken Language Systems Group*, Laboratory for Computer Science, MIT, December 1989.

16. Shlomo Kipnis, "Organization of Systems with Bussed Interconnections," *Digital Systems Research Center*, Palo Alto, March 1990.

17. Shlomo Kipnis, "Organization of Systems with Bussed Interconnections," *IBM Almaden Research Center*, Almaden, CA, February 1990.

18. Shlomo Kipnis, "Organization of Systems with Bussed Interconnections," *IBM Yorktown Research Center*, New York, March 1990.

19. Shlomo Kipnis, "Organization of Systems with Bussed Interconnections," *Phillips Laboratories*, Briarcliff Manor, March 1990.

20. Shlomo Kipnis, "Priority Arbitration with Busses," *DARPA Fall 1989 Contractors Meeting*, Arlington, VA, November 1989.

21. David Kranz, "Mul-T: A High-Performance Parallel Lisp." *SIGPLAN '89*, Portland, Oregon, June 1989.

22. T. Leighton, "Fast Fault-Tolerant Algorithms for Routing on Multibutterflies and Nonblocking Networks," *Carnegie Mellon Computer Science Department*, Pittsburgh, December 1989.

23. T. Leighton, "Fast Fault-Tolerant Algorithms for Routing on Multibutterflies and Nonblocking Networks," *IBM Thomas J. Watson Computer Science Group*, New York, October 1989.

24. T. Leighton, "Fast Fault-Tolerant Algorithms for Routing on Multibutterflies and Nonblocking Networks," *MIT LIDS*, Cambridge, February 1990.

25. T. Leighton, "Fast Fault-Tolerant Algorithms for Routing on Multibutterflies and Nonblocking Networks," *NECUSE Annual Meeting at Amherst*, January 1990.

26. Bruce Maggs, "Fault-Tolerant Routing Algorithms for Multibutterfly Networks," *Duke University*, February 1990.

27. Bruce Maggs, "Fault-Tolerant Routing Algorithms for Multibutterfly Networks," *NEC Research Institute*, Princeton, January 1990.

28. Bruce Maggs, "Fault-Tolerant Routing Algorithms for Multibutterfly Networks," *Rice University*, February 1990.

29. Bruce Maggs, "Fault-Tolerant Routing Algorithms for Multibutterfly Networks," *Bell Communications Research*, Morristown, NJ, February 1990.

30. Bruce Maggs, "Fault-Tolerant Routing Algorithms for Multibutterfly Networks," *IBM Thomas J. Watson Research Center*, New York, February 1990.

31. Bruce Maggs, "Expanders might be practical: Fast algorithms for routing around faults on multibutterflies," *30th Annual Symposium on Foundations of Computer Science*, Triangle Park, October 1989.

32. Keith Nabors, "A Fast Multipole Algorithm for Complex 3-D Geometries," *MIT CAD Review*, Cambridge, October 1989.

33. James Park, "Selection and Sorting in Totally Monotone Arrays," *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, January 1990.

34. James Park, "The Monge Array — An Abstraction and Its Appications," *Ohio State University*, Columbus, February 1990.

35. James Park, "The Monge Array — An Abstraction and Its Appications," *Sandia National Laboratories*, Albuquerque, March 1990.

36. Mark Reichelt, "Waveform Relaxation for Transient Simulation of Two-Dimensional MOS Devices," *Int. Conf. on Computer-Aided Design*, Santa Clara, California, October 1989.

37. Mark Reichelt, "Waveform Relaxation for Transient Simulation of Two-Dimensional MOS Devices," *MIT CAD Review*, Cambridge, October 1989.

38. Jacob White, "A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries," *CAD Seminar*, Berkeley, 1989.

39. Jacob White, "A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries," *U. C. Berkeley*, October 1989.

40. Jacob White, "A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries," *Analog Workshop*, San Jose, January 1990.

41. Jacob White, "A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries," *Digital Equip. Corp. Technical Seminar*, Hudson, MA, March 1990.

42. Jacob White, "A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries," *DARPA VLSI Contractors Meeting*, Wash., D.C., November 1989.

43. Jacob White, "Distortion Analysis of Switched-Capacitor Filters," *MIT CAD Review*, Cambridge, October 1989.

44. Jacob White, "Distortion Analysis of Switched-Capacitor Filters," *Mixed Signal Workshop*, Portland, Oregon, October 1989.

45. Jacob White, "Multipole-Accelerated Boundary Element Methods," *Hydrodynamic Seminar*, MIT Ocean Engineering Dept., February 1990.

46. Jacob White, "Multipole-Accelerated Boundary Element Methods," *Numerical Analysis Seminar*, MIT Math Dept., December 1989.

47. Jacob White, "Numerical Simulation of Switching Power and Filter Circuits," *Microelectronics Consortium of North Carolina Industrial Review*, MCNC, Research Triangle Park, November 1989.

48. Jacob White, "Numerical Simulation of Switching Power and Filter Circuits," *CAD Seminar*, Carnegie Mellon University, March 1990.

49. Jacob White, "Waveform Relaxation for Device Transient Simulation," Invited Talk, *VLSI Review*, MIT, December 1989.

50. Jacob White, "Waveform Relaxation for Device Transient Simulation," Invited Talk, *Sandia National Laboratory*, Livermore, CA, November 1989.

51. Jacob White, "Multipole-Accelerated Boundary Element Methods," *IBM Techical Seminar*, Yorktown, N. Y., March 1990.

9 Selected Publications

1. Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. To appear in *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

2. S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a nonblocking network. To appear in *Proc. 22nd JACM Symp. on Theory of Computing*, May 1990.

3. Srinivas Devadas. Minimization of Functions with Multiple-Valued Outputs: Theory and Applications. To appear in *Proceedings of the 20th Int'l Symposium on Multiple-Valued Logic*, May 1990.

4. K. Nabors, J. White. A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries. In *Proceedings Custom Int. Circuits Conf.*, May, 1989. Updated December, 1989.

APRIL: A Processor Architecture for Multiprocessing

Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Processors in large-scale multiprocessors must be able to tolerate large communication latencies and synchronization delays. This paper describes the architecture of a rapid-context-switching processor called APRIL with support for fine-grain threads and synchronization. APRIL achieves high single-thread performance and supports virtual dynamic threads. A commercial RISC-based implementation of APRIL and a runtime software system that can switch contexts in about 10 cycles is described. Measurements taken for several parallel applications on an APRIL simulator show that the overhead for supporting parallel tasks based on *futures* is reduced by a factor of two over a corresponding implementation on the Encore Multimax. The scalability of a multiprocessor based on APRIL is explored using a performance model. We show that the SPARC-based implementation of APRIL can achieve close to 80% processor utilization with as few as three resident threads per processor in a large-scale cache-based machine with an average base network latency of 55 cycles.

1 Introduction

The requirements placed on a processor in a large-scale multiprocessing environment are different from those in a uniprocessing setting. A processor in a parallel machine must be able to tolerate high memory latencies and handle process synchronization efficiently [3]. Further, memory access times and synchronization overhead will increase as machines are scaled up.

Parallel applications impose processing and communication bandwidth demands on the parallel machine. An efficient and cost-effective machine design achieves a balance between the processing power and the communication bandwidth the machine provides. An imbalance is created when an underutilized processor cannot fully exploit the available network bandwidth. When the network has bandwidth to spare, low processor utilization can result due to high network latency. An efficient processor design for multiprocessors provides a means for hiding latency. Provided that sufficient parallelism exists, a processor that rapidly switches to an alternate thread of computation during a remote memory request can achieve high utilization.

Processor utilization also diminishes due to synchronization latency. Spin lock accesses have a low overhead of memory requests, but busy-waiting on a synchronization event wastes processor cycles. Synchronization mechanisms that avoid busy-waiting by means of process blocking incur a high overhead, and are typically associated with coarse-grain objects.

Full-empty bit synchronization [27] in a rapid context switching processor allows more efficient fine-grain synchronization. This scheme associates synchronization information with objects at the granularity of a data word, allowing a low-overhead expression of maximum concurrency. Because the processor can rapidly switch to other threads, wasteful iterations in spin-wait loops are interleaved with useful work from other threads. This drastically reduces the effects of synchronization on processor utilization. Dataflow computing tries to achieve the same effects using I-structures [4].

This paper describes the architecture of APRIL, a processor designed for large-scale multiprocessing. APRIL builds upon previous research on processors for parallel architectures such as HEP [27], SPUR [8], MASA [11], P-RISC [21], [15], and [16]. Most of these processors support *fine-grain interleaving* of instruction streams from multiple processes, but suffer from poor single-thread performance. APRIL does not support cycle-by-cycle interleaving of threads; instead APRIL executes instructions from a given thread until it performs a remote memory request or fails in a synchronization attempt. The use of caches minimizes the need for remote memory operations. We show that such *coarse-grain multithreading* allows a simple processor design with context switch overheads in the range of 4-10 cycles, without significantly hurting overall system performance (although the pipeline design is complicated by the need to handle pipeline dependencies). In APRIL, thread scheduling is done in software, and unlimited virtual dynamic threads are supported. APRIL supports *full-empty* bit synchronization [27], and provides tag support for `futures` [12].

By taking a systems-level design approach that considers not only the processor, but also the compiler and runtime system, we were able to migrate several non-performance-critical operations into the software system, greatly simplifying the processor design. In fact, APRIL's simplicity allows an implementation based on minor modifications to an existing RISC processor design. We describe such an implementation based on Sun Microsystem's SPARC processor [1]. A compiler for APRIL, a runtime system, and an APRIL simulator are operational. We present simulation results for several parallel applications on APRIL's efficiency in handling fine-grain threads and assess the scalability of multiprocessors based on a coarse-grain multithreaded processor using an analytical model. Our SPARC-based processor supports four hardware contexts and can switch contexts in about 10 cycles, which yields roughly 80% processor utilization in a system with an average base network latency of 55 cycles.

The rest of this paper is organized as follows. Section 2 overviews our multiprocessor system architecture. The programming model is discussed in Section 3. The architecture of APRIL is discussed in Section 4, and its instruction set is described in Section 5. A SPARC-based implementation of APRIL is detailed in 6. Section 7 discusses the implementation and performance of the APRIL runtime system. Performance measurements of APRIL based on simulations are presented in Section 8. We evaluate the scalability of multithreaded processors in Section 9.

2 The ALEWIFE System

APRIL is the processing element of ALEWIFE, a large-scale multiprocessor being designed at MIT. ALEWIFE is a cache-coherent machine with distributed, globally-shared memory. Cache coherence is maintained using a message-based directory protocol [6] over a low-dimension direct network [24]. The directory is distributed with the processing nodes.

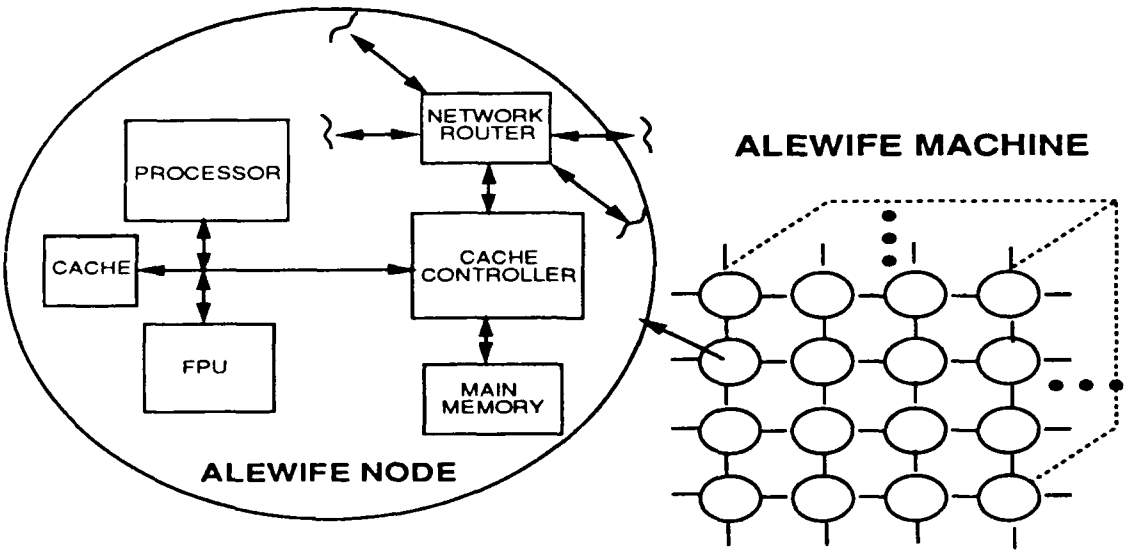

Figure 1: ALEWIFE node

As shown in Figure 1, each ALEWIFE node consists of a processing element, floating-point unit, cache, main memory, cache/directory controller and a network routing switch. Multiple nodes are connected via a direct, packet-switched network.

The controller synthesizes a global shared memory space via messages to other nodes, and satisfies requests from other nodes directed at local memory. It maintains strong cache coherence [7] for memory accesses, but provides special mechanisms for bypassing coherence when appropriate. On exception conditions, such as cache misses and failed synchronization attempts, the controller can choose to trap the processor or to make the processor wait. A rapid-context-switching processor reduces the ill effects of long latency acknowledgments resulting from a strong cache coherence protocol. To allow experimentation with other programming models, the controller provides support for weaker forms of coherence and facilities for "nonstandard" forms of communication, such as preemptive interprocessor interrupts and block transfers. Special controller functions may be exploited by the runtime system through the use of APRIL instructions.

The ALEWIFE system uses a low-dimension direct network. Such networks scale easily and maintain high nearest-neighbor bandwidth. However, the longer expected latencies of a low-dimension direct network over those of indirect multistage networks increase the need for processors that can tolerate long latencies. Furthermore, the lower bandwidth of direct networks over that of indirect networks with the same channel width introduce interesting tradeoffs in the design of multithreaded processors. These are considered in Section 9 and in more detail in [2].

In the ALEWIFE system, a context switch occurs whenever the network must be used to satisfy a request. A context switch can also be forced on a failed synchronization attempt. Since caches reduce the network request rate, we can employ coarse-grain multithreading (context switch every 50-100 cycles) instead of fine-grain multithreading (context switch every cycle). This simplifies processor design considerably because context switches can be more expensive (4

to 10 cycles), and it allows migration of functionality such as scheduling into runtime software. Single-thread performance is optimized, and well-known techniques used in RISC processors for enhancing pipeline performance can be applied [13]. Custom design of a processing element is not required in the ALEWIFE system; indeed, we are using a modified version of a commercial RISC processor for our first-round implementation.

3 Programming Model

Our experimental programming language for ALEWIFE is Mul-T [18], an extended version of Scheme [23]. Mul-T's basic mechanism for generating concurrent tasks is the **future** construct. The expression (**future** X), where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *future* to eventually hold the value of X. When created, the future is in an *unresolved*, or *undetermined*, state. When the value of X becomes known, the future *resolves* to that value, effectively mutating into the value of X and losing its identity as a future. Concurrency arises because the expression (**future** X) returns the future as its value without waiting for the future to resolve. Thus, the computation containing (**future** X) can proceed concurrently with the evaluation of X. All tasks execute in a shared address-space.

The result of supplying a future as an operand of some operation depends on the nature of the operation. *Non-strict* operations, such as passing a parameter to a procedure, returning a result from a procedure, assigning a value to a variable, and storing a value into a field of a data structure, can use a future as easily as any other kind of value, and take no special note of futures. *Strict* operations such as addition and comparison, if applied to an unresolved future, are suspended until the future resolves and then proceed, using the value to which the future resolved as though that had been the original operand.

The act of suspending if an object is an unresolved future and then proceeding when the future resolves is known as *touching* the object. The touches that automatically occur when strict operations are attempted are referred to as *implicit touches*. Mul-T also includes an *explicit* touching or "strict" primitive (**touch** X) that touches the value of the expression X and then returns that value.

Futures can only express control-level parallelism. In a large class of algorithms, data parallelism is more appropriate. Barriers are a useful means of synchronization for such applications on MIMD machines, but force unnecessary serialization. The same serialization occurs in SIMD machines. Implementing data-level parallelism in a MIMD machine that allows the expression of maximum concurrency requires cheap fine-grain synchronization associated with each data object. We provide this support in hardware with *full/empty bits*. Mul-T is being augmented with constructs for data-level parallelism. We are also extending Mul-T with primitives for software cache coherence, placement of data, and task scheduling. As an example, the programmer can use **future-on** which works just like a normal **future** but allows the specification of the node on which to schedule the future. Extending Mul-T in this way allows us to experiment with techniques for enhancing locality and to research language-level issues for programming parallel machines.

4 Processor Architecture

APRIL is a pipelined RISC processor extended with special mechanisms for multiprocessing. This section overviews the architecture of APRIL focusing on the features of APRIL that support multithreading, fine-grain synchronization, cheap futures, and other models of computation.

4.1 Overview

The left half of Figure 2 depicts the user-visible processor state comprising four sets of general purpose registers, and four sets of Program Counter (PC) chains and Processor State Registers (PSR). The PC chain represents the instruction addresses corresponding to a thread, and the PSR holds various pieces of process-specific state such as the condition codes (CCs). In this paper the terms process, thread, context, and task are used equivalently. Each register set together with a single PC-chain and PSR is conceptually grouped into a single entity called a *task frame* (using terminology from [11]). Only one task frame is active at a given time and is designated by a current frame pointer (FP). All register accesses are made to the active register set and instructions are fetched using the active PC-chain. Additionally, a set of 8 global registers that are always accessible regardless of the FP is provided.

Registers are 32 bits wide. The PSR is also a 32 bit register and can be read into and written from the general registers. Special instructions can read and write the FP register. The PC-chain includes the Program Counter (PC) and next Program Counter (nPC) which are 32 bit registers and are not directly accessible. The above assumes a single cycle branch delay slot. Condition codes are set as a side effect of compute instructions. A longer branch delay might be necessary if the branch instruction itself does a compare [9]; in this case the PC chain is correspondingly longer. Words in memory are 32 bits wide, and have an additional synchronization bit called the *full/empty* bit.

Use of multiple register sets on the processor, as in the HEP, allows rapid context switching. A fast context switch is achieved by changing the frame pointer and emptying the pipeline. The cache controller forces a context switch on the processor, typically on remote network requests, and on certain unsuccessful full/empty bit synchronizations. APRIL implements *futures* using the trap mechanism. For our proposed experimental implementation based on SPARC, which does not have four separate PC and PSR frames, context switches are also caused through traps. Therefore, a fast trap mechanism is essential. When a trap is signalled in APRIL, the trap state machine lets the pipeline empty, and passes control to the trap handler. The trap hardware also saves the PC-chain into the general registers. The trap handler executes in the same task frame as the thread that trapped so that it can access all its registers.

4.2 Coarse-Grain Multithreading

In most processor designs to date (e.g. [11, 27, 21, 16]), multithreading has involved cycle-by-cycle interleaving of threads. Such fine-grain multithreading has been used to hide memory latency and also to achieve high pipeline utilization. By maintaining instructions from different threads in the pipeline at any instant pipeline dependencies are avoided, but at the price of poor single-thread performance.

In the ALEWIFE machine, we are primarily concerned with the large latencies associated

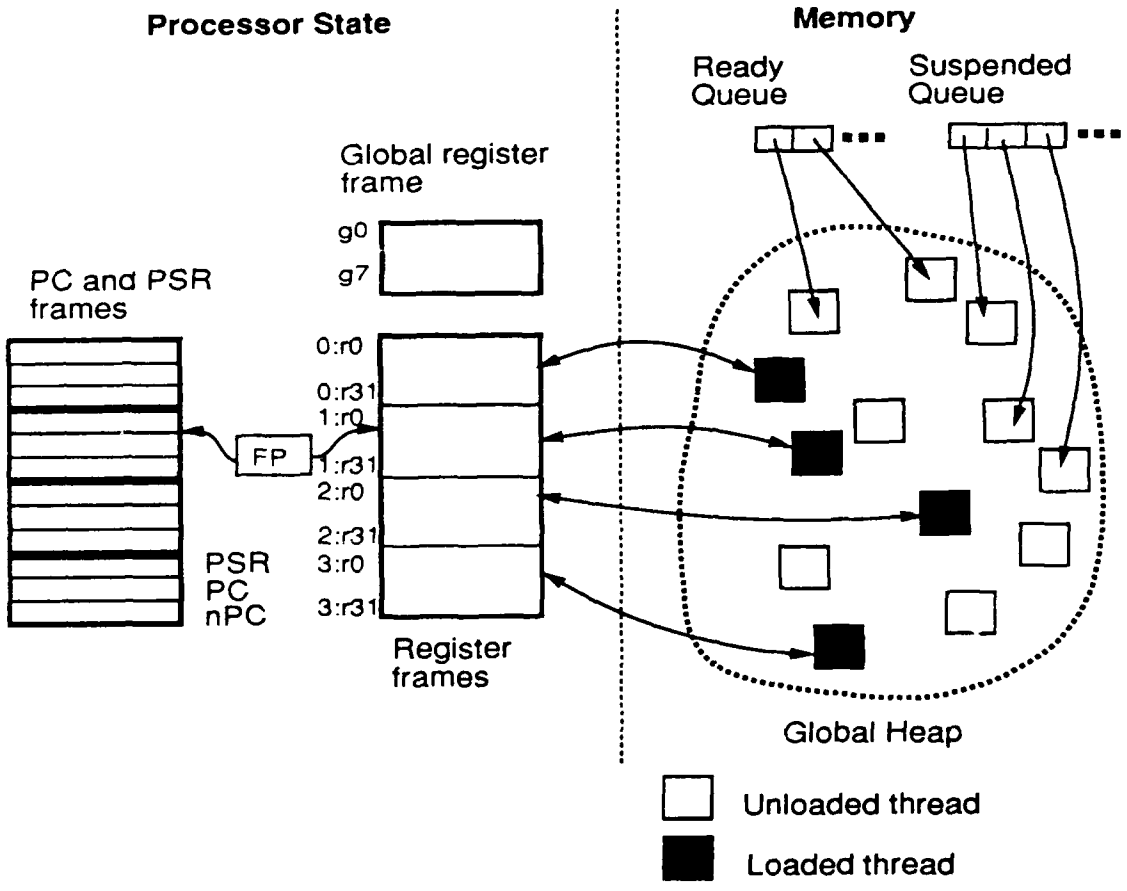

Figure 2: Processor State and Virtual Threads

with cache misses that require a network access. Good single thread performance is also important. Therefore the processor continues executing a single thread until a memory operation involving a remote request (or an unsuccessful synchronization attempt) is encountered. The controller forces the processor to switch to another thread, while it services the request. This approach is called *coarse-grain multithreading*. Processors in message passing multicomputers [25, 31, 10, 5] have traditionally taken this approach to allow overlapping of communication with computation.

Context switching in APRIL is achieved by changing the frame pointer. Since APRIL has four task frames, it can have up to four threads loaded on it. The thread that is being executed by APRIL resides in the task frame pointed to by the FP. A context switch simply involves letting the processor pipeline empty while saving the PC-chain and then changing the FP to point to another task frame.

Threads in ALEWIFE live in a virtual namespace. Only a small subset of all threads can be physically resident on the processors; these threads are called *loaded threads*. The remaining threads are refered to as *unloaded threads* and live on various queues in memory, waiting their turn to be loaded. In a sense, the set of task frames act like a cache on the virtual threads. This organization is illustrated in Figure 2. The scheduler tries to choose threads from this set of loaded threads for execution to minimize the overhead of saving and restoring threads into memory. When control eventually passes back to the thread that suffered a remote request, the controller should have completed servicing the request, provided the other threads ran for enough cycles. In other words, remote memory latencies can be completely overlapped if the product of the number of loaded threads and the average switch time is greater than the average time to service a request. By maximizing local cache and memory accesses, the need for context switching reduces to once every 50 or 100 cycles, which allows us to tolerate latencies in the range of 150 to 300 cycles with 4 task frames (see Section 9).

Rapid context switching is used to hide the latency encountered in several other trap events, such as synchronization faults (or attempts to load from "empty" locations). These events can either cause the processor to suspend execution (wait) or to take a trap. In the former case, the controller holds the processor until the request is satisfied. This typically happens on local memory cache misses, and on certain full/empty bit tests. If a trap is taken, the trap handling routine can respond by:

1. *spinning* – immediately return from the trap and retry the trapping instruction.
2. *switch spining* – context switch without unloading the trapped thread.
3. *blocking* – unload the thread.

The above alternatives must be considered with care, for incorrect choices can create or exacerbate starvation and thrashing problems. An extreme example of starvation is this: all loaded threads are spinning or switch spining on an exception condition that an unloaded thread is responsible for fulfilling. We are investigating several possible mechanisms to handle such problems, including a special controller initiated trap on certain failed synchronization tests, whose handler unloads the thread.

An important aspect of the ALEWIFE system is its combination of caches and multithreading. While this combination is advantageous, it also creates a unique class of thrashing and

starvation problems. For example, forward progress can be halted if a context executing on one processor is writing to a location while a context on another processor is reading from it. These two contexts can easily play "cache tag", since writes to a location force a context switch and invalidation of other cached copies, while reads force a context switch and transform read-write copies into read-only copies. Another problem involves thrashing between an instruction and its data; a context will be blocked if it has a load instruction mapped to the same cache line as the target of the load. These and related problems have been addressed via appropriate hardware interlock mechanisms.

4.3 Support for Futures

Executing a Mul-T program with futures incurs two types of overhead not present in sequential programs:

1. Strict operations must check their operands for availability before using them.
2. There is a cost associated with creating new threads.

Detection of Futures Operand checks for futures done in software imply wasted cycles on every strict operation. Our measurements with Mul-T running on an Encore Multimax shows that this is expensive. Even with clever compiler optimizations there is close to a factor of two loss in performance over a purely sequential implementation (see Table 3). Our solution employs a tagging scheme with hardware-generated traps if an operand to a strict operator is a future. We believe that this hardware support is necessary to make futures a viable construct for expressing parallelism. From an architectural perspective, this mechanism is similar to dynamic type checking in Lisp. However, this mechanism is necessary even in a statically typed language in the presence of the dynamic futures.

APRIL uses a simple data type encoding scheme for automatically generating a trap when operands to strict operators are futures. Implementation details are discussed in Section 6. This obviates the need to explicitly inspect in software the operands to every compute instruction. This is important because we do not want to hurt the efficiency of all compute instructions because of the possibility an operand is a future.

Lazy Task Creation Little can be done to reduce the cost of task creation if *future* in a program is taken as a command to create a new task. In many programs the possibility of creating an excessive number of fine-grain tasks exists. On the other hand, given the semantics of Mul-T, failure to create a new task could result in deadlock. These observations lead us to the notion of *lazy futures* [20]. With lazy futures a *future* expression does not create a new task, but simply treats the expression as a procedure call, leaving behind a placeholder where the new task could have been created. The new task is created only when some processor becomes idle and looks for work. Thus, the user can specify the maximum possible parallelism, without the overhead of creating all the tasks. parallel task creation is paid only for tasks that actually run on a different processor.

Lazy futures, in a sense, are a means of dynamically partitioning programs to try to achieve just the right amount of parallelism.

Lazy futures introduce a race condition between a processor looking for a new task to create and a processor trying to avoid creating the new task by executing the future as a procedure call. APRIL's fine-grain synchronization implements lazy futures efficiently.

4.4 Fine-grain synchronization

Besides support for lazy futures, efficient fine-grain synchronization is essential for large-scale parallel computing. Both the dataflow and data-parallel models of computation rely heavily on the availability of cheap fine-grain synchronization. The unnecessary serialization imposed by barriers in MIMD implementations of data-parallellism can be avoided by allowing fine-grain word-level synchronization in data structures. The traditional **test&set** based synchronization requires extra memory operations and separate data storage for the lock and for the associated data. Busy-waiting or blocking in conventional processors waste additional processor cycles.

APRIL adopts the full/empty bit approach used in the HEP [27] to reduce both the storage requirements and the number of memory accesses. A bit associated with each memory word indicates the state of the word: full or empty. The load of an empty location or the store into a full location traps the processor causing a context switch, which helps hide synchronization delay. Traps also obviate the additional software tests of the lock in **test&set** operations. A similar mechanism is used to implement I-structures in dataflow machines [4], however APRIL is different in that it implements such synchronizations through traps handled in software.

4.5 Mechanisms for Multimodel Support

ALEWIFE is primarily an experimental shared-memory multiprocessor with strongly coherent caches. However, we are considering several additional mechanisms which will permit explicit management of caches and efficient use of network bandwidth. These mechanisms present different computational models to the programmer. In general, use of such mechanisms requires "intelligent" software and sophisticated compilers.

First on this list is support for explicit cache management. We have loads and stores that bypass the hardware coherence mechanism, and a *flush* operation that permits software writeback and invalidation of cache lines. A loaded context has a *fence counter* that is incremented for each dirty cache line that is flushed and decremented for each acknowledgement from memory. This fence counter may be examined to determine if all writebacks have completed. Such software coherence is along the lines of that considered by Shasha and Snir [26]. Cache coherence is an active area of research and providing an option for hardware or software cache coherence in ALEWIFE will let us study this issue.

We are also proposing a block-transfer mechanism for efficient transfer of large blocks of data. The block-transfer mechanism examines caches only on source and destination nodes, bypassing hardware cache coherence. Fence counters are used for synchronization here also.

Finally, we are considering an interprocessor-interrupt mechanism (IPI) which permits preemptive messages to be sent to specific processors. Each IPI will have an interrupt level and up to five words of data. IPIs offer reasonable alternatives to polling and, in conjunction with block-transfers, form a primitive for the message-passing computational model.

Although each of these mechanisms adds complexity to our cache controller, they are easily

Type	Format	Data movement	Control flow
Compute	`op s1 s2 d`	`d ← s1 op s2`	`PC+1`
Memory	`ld` *type* `a d`	`d ← mem[a]`	`PC+1`
	`st` *type* `d s`	`mem[a] ← s`	`PC+1`
Branch	`j`*cond* `offset`		`if` *cond* `PC+offset else PC+1`
	`jmpl offset d`	`d ← PC`	`PC+offset`

Table 1: Basic instruction set summary.

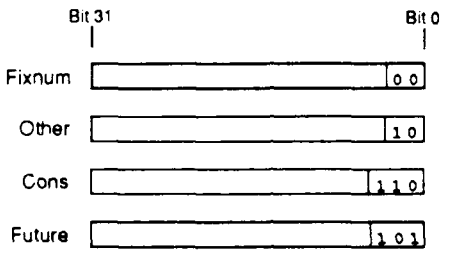

Figure 3: Data Type Encodings

implemented in the processor through "out-of-band" instructions as discussed in Section 6.3. Since an overriding design consideration for ALEWIFE is simplicity, we are evaluating the utility of these mechanisms carefully.

5 Instruction Set

APRIL has a basic RISC instruction set augmented with special memory instructions for full/empty bit operations, multithreading, and cache support. The attraction of an implementation based on simple SPARC processor modifications has resulted in a basic SPARC-like instruction set [1]. All registers are addressed relative to a current frame pointer. Compute instructions are 3-address register-to-register arithmetic/logic operations. Conditional branch instructions take an immediate operand and conditionally increment the PC by the value of the immediate operand depending on the condition codes set by the arithmetic/logic operations. Memory instructions move data between memory and the registers, and also interact with the cache and the full/empty bits. The basic instruction categories are summarized in Table 1. The remainder of this section describes features of APRIL instructions used for supporting multiprocessing.

Data Type Formats APRIL supports tagged pointers for Mul-T by encoding the pointer type in the low order bits of a data word. Figure 3 lists the different type encodings. An important purpose of this type encoding scheme is to support hardware detection of futures.

Name	Type	Reset f/e bit	EL[1] trap	CM[2] response
ldtt	1	No	Yes	Trap
ldett	2	Yes	Yes	Trap
ldnt	3	No	No	Trap
ldent	4	Yes	No	Trap
ldnw	5	No	No	Wait
ldenw	6	Yes	No	Wait
ldtw	7	No	Yes	Wait
ldetw	8	Yes	Yes	Wait

[1]Empty location. [2]Cache miss.

Table 2: Load Instructions.

Future Detection and Compute Instructions A compute instruction is a *strict* operation: special action has to be taken if either of its operands is a future. APRIL generates a trap if a future is encountered by a compute instruction. From the encodings presented in Figure 3, future pointers are easily detected by their non-zero least significant bit.

Memory Instructions Memory instructions are a complex category because they interact with the full/empty bits and the cache controller. On a memory access, two data exceptions can occur: the accessed location may not be in the cache (a cache miss), and the accessed location may be empty on a load or full on a store (a full/empty exception). Depending on the specific memory instruction executed by the processor, on a cache miss, the cache/directory controller can trap the processor or make the processor wait until the data is available. On full/empty exceptions, the controller can trap the processor, or allow the processor to continue execution. Load instructions also have the option of setting the full/empty bit of the accessed location to empty while store instructions have the option of setting the bit to full. These options give rise to 8 kinds of loads and 8 kinds of stores. The load instructions are listed in Table 2. Store instructions are similar except that they trap on full locations instead of empty locations.

A memory instruction also shares responsibility for detecting futures in either of its address operands. Like compute instructions, memory instructions also trap if the least significant bit of either of their address operands are non-zero. This introduces the restriction that objects in memory cannot be allocated at byte boundaries. This, however, is not a problem because object allocation at word boundaries is favored for other reasons [14]. This trap provides support for implicit future touches in operators that dereference pointers like **car**.

Full/Empty Bit Conditional Branch Instructions The non-trapping memory instructions allow testing of the full/empty bit by setting a condition bit indicating the state of the full/empty bit associated with the memory word. APRIL provides conditional branch instructions, `Jfull` and `Jempty`, that dispatch on this condition bit. This provides a way to explicitly control the action taken following a memory instruction that would normally trap on a full/empty exception.

Frame Pointer Instructions Instructions are provided for manipulating the register frame pointer (FP). FP points to the register frame on which the currently executing thread resides. An

INCFP instruction increments the FP to point to the next task frame while a DECFP instruction decrements it. The incrementing and decrementing is done modulo the number of task frames. RDFP reads the value of the FP into a register and STFP writes the contents of a register into the FP. These instructions allow the user to have total control over the value of the FP.

Instructions for Other Mechanisms The special mechanisms discussed in Section 4.5 are made available through "out-of-band" instructions. APRIL provides several such instructions, including a FLUSH instruction and software coherent versions of the load-store instructions discussed above. Interprocessor-interrupts, block-transfers, and FENCE operations are initiated via memory-mapped I/O instructions (LDIO, STIO).

6 An Implementation of APRIL

An ALEWIFE node consists of several interacting subsystems: processor, floating-point unit, cache, memory, cache and directory controller, and network controller. For the first round implementation of the ALEWIFE system, we plan to use a modified SPARC processor and an unmodified SPARC floating-point unit.[1] There are several reasons for this choice. First, we have chosen to devote our limited resources to the design of a custom ALEWIFE cache and directory controller rather than to processor design. Second, the register windows in the SPARC processor permit a simple implementation of course-grain multithreading. Third, most of the instructions envisioned for the original APRIL processor map directly to single or double instruction sequences on the SPARC. Software compatibility with a commercial processor allows easy access to a large body of software. Furthermore, use of a standard processor permits us to ride the technology curve; we can take advantage of new technology as it appears.

6.1 Rapid Context Switching on SPARC

SPARC processors contain an implementation-dependent number of overlapping register windows meant for speeding up procedure calls [17]. The current register window is altered via special SPARC instructions (SAVE and RESTORE) that modify the Current Window Pointer (CWP), typically during the procedure call and return sequences. Furthermore, traps increment the CWP, while the trap return instruction (RETT) decrements this pointer.

SPARC's register windows are suited for rapid context switching because most of the state of a process (i.e. its 24 local registers) can be switched with a single-cycle instruction. Although we are not using multiple register windows within a single thread, this should not significantly hurt performance [29, 28]. Register windows also permit rapid trap handling.

To implement course-grain multithreading, we employ two register windows per task frame – a user window and a trap window. The SPARC processor chosen for our implementation has eight register windows, allowing a maximum of four hardware task frames. As discussed in Section 9, four appear to be sufficient. Since the SPARC does not have multiple program counter (PC) chains and processor status registers (PSR), our trap code must explicitly save and restore the PSRs during context switches (the PC chain is saved by the trap itself). These

[1] The SPARC-based implementation effort is in collaboration with LSI Logic Corporation.

values are saved in the trap window. Because the SPARC has a minimum trap overhead of four to five cycles (for squashing the pipeline and computing the trap vector), context switches will take at least this long. See section 7.1 for further information.

The SPARC floating-point unit does not support register windows, but has a single, 32 word register file. To retain rapid context switching ability for applications that require efficient floating point, we have divided the floating point register file into four sets of eight registers. This is achieved by modifying floating-point instructions in a context dependent fashion as they are loaded into the FPU and by maintaining four different sets of condition bits. A modification of the SPARC processor will make the CWP available externally to allow insertion into the FPU instruction.

6.2 Support for Futures

We detect futures on the SPARC via two separate mechanisms. Future pointers are tagged in such a way that their lowest bit is set. Thus, direct use of a future pointer is flagged with a *word-alignment trap*. Further, a strict operation, such as subtraction, applied to one or more future pointers is flagged with a modified *non-fixnum trap*, which is triggered if one of the operands has its lowest bit set (as opposed to one of the lowest *two* bits, as in the SPARC specification).

6.3 Implementation of Loads and Stores

The SPARC definition includes the Alternate Space Indicator (ASI) feature that permits a simple implementation of APRIL's many load and store instructions (described in Section 5). The ASI is available externally as an eight-bit field. Normal memory accesses use four of the 256 ASI values to indicate user/supervisor and instruction/data accesses. Special SPARC load and store instructions (`LDASI` and `STASI`) permit use of the other 252 ASI values. Our first-round implementation uses different ASI values to distinguish between flavors of load and store instructions, special mechanisms, and I/O instructions.

6.4 Interaction with the Cache Controller

The cache controller in the ALEWIFE system maintains strong cache coherence, performs full/empty bit synchronization, and implements special mechanisms. By examining the processor's ASI bits during memory accesses, it can select between different load/store and synchronization behavior, and can determine if special mechanisms should be employed. Through use of the Memory Exception (MEXC) line on SPARC, it can invoke synchronous traps corresponding to cache misses and synchronization (full/empty) mismatches. The controller can suspend processor execution using the MHOLD line. It passes condition information to the processor through the Coprocessor Condition bits (CCCs), permitting the full/empty conditional branch instructions (`Jfull` and `Jempty`) to be implemented as coprocessor branch instructions. Asynchronous traps (IPI's) are delivered via the SPARC's asynchronous trap lines.

7 Compiler and Run-Time System

The compiler and run-time system are integral parts of the processor design effort. A Mul-T compiler for APRIL and a run-time system written partly in APRIL assembly code and partly in T [22] have been implemented. Constructs for user-directed placement of data and processes have also been implemented. The run-time system includes the trap and system callable routines, Mul-T run-time support, a scheduler, and a system boot routine.

Since a large portion of the support for multithreading, fine-grained synchronization and futures is provided in software through traps and run-time routines, trap handling must be fast. Below, we briefly describe the implementation and performance of the routines used for trap handling and context switching.

7.1 Cache Miss and Full/Empty Traps

Cache miss traps occur on cache misses that require a network request and cause the processor to context switch. Full/empty synchronization exceptions can occur on certain memory instructions described in Section 5. The processor can respond to these exceptions by spin waiting, spin blocking, or blocking the thread. In our current implementation, traps handle these exceptions by spin blocking, which involve a context switch to the next task frame.

In our SPARC-based design of APRIL, we implement context switching through the trap mechanism using instructions that change the task frame pointer FP. The following is a trap routine that context switches to the thread in the next task frame.

```
rdpsr psrreg ; save PSR into a reserved reg.
save         ; increment the window pointer
save         ; by 2
wrpsr psrreg ; restore PSR for the new context
jmpl  r17    ; return from trap and
rett  r18    ; reexecute trapping instruction
```

We count 5 cycles for the trap mechanism to allow the pipeline to empty and save relevant processor state before passing control to the trap handler. The above trap handler takes an additional 6 cycles for a total of 11 cycles to effect the context switch. In a custom APRIL implementation, the cycles lost due to PC saves in the hardware trap sequence, and those in calling the trap handler for the PSR saves/restores and double incrementing the frame pointer can be obviated, allowing a four-cycle context switch.

7.2 Future Touch Trap

A future touch trap can be signalled by compute and memory instructions. When this occurs, the future that caused the trap will be found in a register because APRIL uses a load/store architecture. Ideally, the trap routine should be provided enough information to allow easy identification of this register. In our implementation however, the trap handler has to decode the trapping instruction to find that register, which makes future touch traps more expensive than necessary. Once it has found the future, the trap handler has to determine if the future

has been resolved by looking at the full/empty bit of the future's value slot. If it is resolved, the future in the register is replaced with the resolved value; otherwise the trap routine can decide to spin block or block the thread that trapped. Currently, our future touch trap handler takes 23 cycles to execute if the future is resolved.

If the trap handler decides to block the thread on an unresolved future, the thread must be unloaded from the hardware task frame, and an alternate thread may be loaded. Loading a thread involves writing the state of the thread, including its general registers, its PC chain, and its PSR, into a hardware task frame on the processor, and unloading a thread involves saving the state of a thread out to memory. Loading and unloading threads are expensive operations unless there is special hardware support for block movement of data between registers and memory. Because the scheduling mechanism favors processor-resident threads, loading and unloading of threads is infrequent. However, this is an issue that is under investigation.

8 Performance Measurements

This section presents some results on APRIL's performance in handling fine-grain tasks. The next section will evaluate the impact of multithreaded processors in large-scale systems using an analytical model.

We have implemented a simulator for the ALEWIFE system called ASIM written in C and T. Figure 4 illustrates the organization of the simulator. The Mul-T compiler produces APRIL code, which gets linked with the run-time system to yield an executable program. The instruction-level APRIL processor simulator interprets APRIL instructions. It is written in T and simulates 40,000 APRIL instructions per second when run on a SPARCStation 330. The processor simulator interacts with the cache and directory simulator (written in C) on memory instructions. The cache simulator in turn interacts with the network simulator (also written in C) whenever it needs to make remote memory operations over the network. The simulator has proved to be a useful tool in evaluating system-wide architectural tradeoffs; it provides more accurate results that a trace driven simulation. The speed of the simulator has allowed us to execute lengthy parallel programs. As an example, in a run of **speech** (described below), the simulated program ran for 100 million simulated cycles before completing.

Evaluation of the ALEWIFE architecture through simulations on ASIM is in progress. A sampling of our results on the performance of APRIL running parallel programs is presented here. Table 3 lists the execution times of four programs written in Mul-T: **fib**, **factor**, **queens** and **speech**. **fib** is the ubiquitous doubly recursive fibonacci program with **future**s around each of its recursive calls, **factor** finds the largest prime factor of each number in a range of numbers and sums them up, **queens** finds all solutions to the n-queens problem for $n = 8$ and **speech** is a modified Viterbi graph search algorithm used a in connected speech recognition system called SUMMIT being developed by the Spoken Language Systems Group at MIT. We ran each program on the Encore Multimax, on APRIL using normal futures and on APRIL using lazy futures. For purposes of comparison, execution time has been normalized to the time taken to execute a sequential version of each program, *i.e.*, with no futures and compiled with the optimizing T-compiler [19].

The difference between running the same sequential code on T and on Mul-T on the Encore Multimax (columns "T seq" and "Mul-T seq") is due to the overhead of future detection because

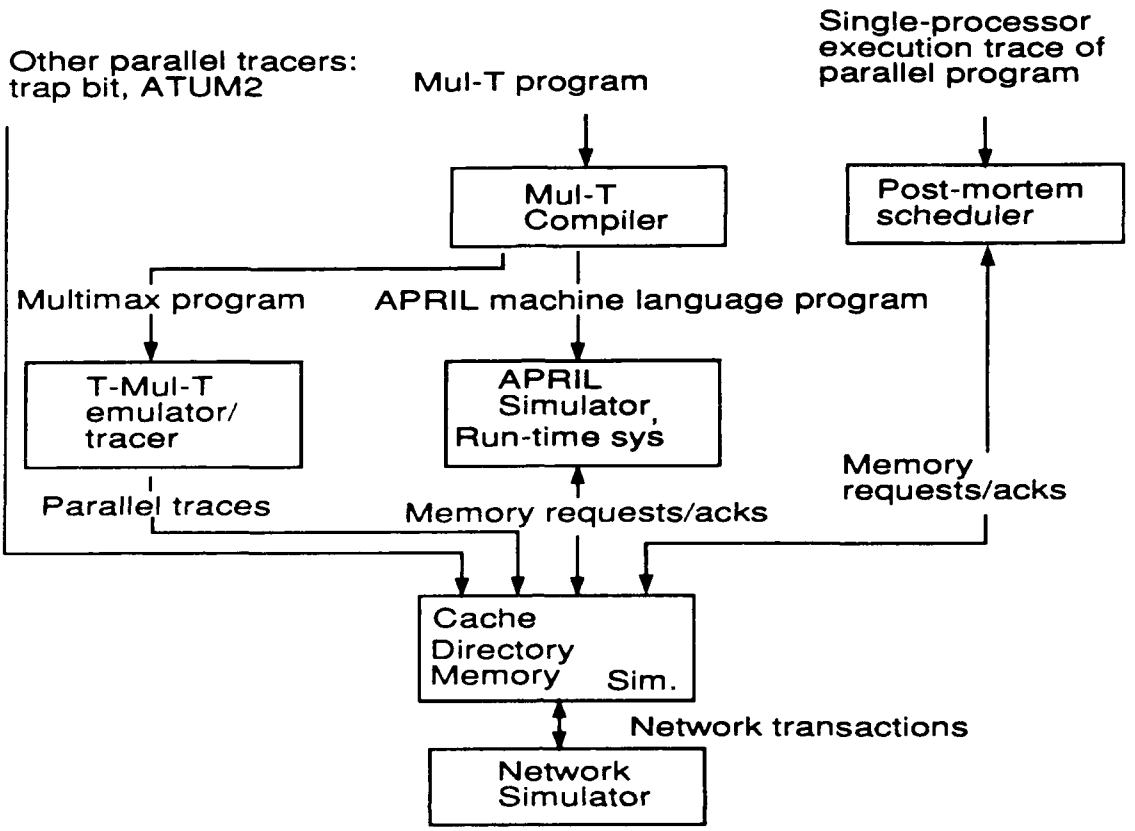

Figure 4: Simulator Organization

		T	Mul-T					
Program	System	seq	seq	1	2	4	8	16
fib	Encore	1.0	1.8	28.9	16.3	9.2	5.1	
	APRIL	1.0	1.0	14.2	7.1	3.6	1.8	0.97
	APR-lazy	1.0	1.0	1.5	0.78	0.44	0.29	0.19
factor	Encore	1.0	1.4	1.9	0.96	0.50	0.26	
	APRIL	1.0	1.0	1.8	0.90	0.45	0.23	0.12
	APR-lazy	1.0	1.0	1.0	0.52	0.26	0.14	0.09
queens	Encore	1.0	1.8	2.1	1.0	0.54	0.31	
	APRIL	1.0	1.0	1.4	0.67	0.33	0.18	0.10
	APR-lazy	1.0	1.0	1.0	0.51	0.26	0.13	0.07
speech	Encore	1.0	2.0	2.3	1.2	0.62	0.36	
	APRIL	1.0	1.0	1.2	0.60	0.31	0.17	0.10
	APR-lazy	1.0	1.0	1.0	0.52	0.27	0.15	0.09

Table 3: Execution time for Mul-T benchmarks. "T seq" is T running sequential code, "Mul-T seq" is Mul-T running sequential code, 1 through 16 denote number of processors running parallel code.

the Encore does not support hardware detection of futures. On the Encore, future detection introduces an overhead of a factor of 2, even though no futures are actually created. There is no overhead on APRIL, which demonstrates the advantage of tag support for futures.

The difference between running sequential code on Mul-T and running parallel code on Mul-T with one processor ("Mul-T seq" and 1) is due to the overhead of thread creation and synchronization in a parallel program. This overhead is very large for the **fib** benchmark on both the Encore and APRIL using normal futures because of very fine-grain thread creation. This overhead accounts for approximately a factor of 28 in execution time, discounting the overhead due to future detection. For APRIL with normal futures, this overhead accounts for a factor of 14. Lazy task creation on APRIL creates threads only when the machine has the resources to execute them, and performs much better because it has the effect of dynamically partitioning the program into coarser-grain threads and creating fewer futures. The overhead introduced is only a factor of 1.5. In all of the programs, APRIL consistently demonstrates lower overhead due to support for thread creation and synchronization over the Encore.

The numbers for multiple processor executions on APRIL (2 – 16) were measured using the processor simulator without the cache and network simulators, in effect simulating a shared-memory machine with a zero memory latency. The numbers demonstrate that APRIL and its run-time system allow parallel program performance to scale when synchronization and task creation overheads are taken into account, but when memory latency is ignored. The effect of communication in large-scale machines depends on several factors such as scheduling, which are active areas of investigation.

9 Scalability of Multithreaded Processor Systems

Multithreading enhances processor efficiency by allowing execution to proceed on alternate computation threads while the memory requests of previous ones are being satisfied. However, any new mechanism is useful only if it enhances *overall system performance*. This section analyzes the system performance of multithreaded processors.

A multithreaded processor design must address the tradeoff between reduced processor idle time and the increased cache miss rates, network contention, and context management overhead. The private working sets of multiple contexts interfere in the cache. The added interference misses coupled with the higher average traffic generated by a fully utilized processor impose higher bandwidth demands on the interconnection network. Context management instructions required to switch the processor between threads also add to the overhead. Furthermore, the application must display sufficient parallelism to allow multiple thread assignment to each processor.

What is a good performance metric to evaluate multithreading? A good measure of system performance is system power, which is the product of the number of processors and the average processor utilization. Provided the computation of processor utilization takes into account the deleterious effects of cache, network, and context-switching overhead, the processor utilization is itself a good measure.

We have developed a model for multithreaded processor utilization that includes the cache, network, and switching overhead effects. A detailed analysis is presented in [2]. This section will summarize the model and our chief results. Processor utilization U as a function of the number

of threads resident on a processor p is derived as a function of the cache miss rate $m(p)$, the network latency $T(p)$, the context switching overhead C:

$$U(p) = \begin{cases} \frac{p}{1+T(p)m(p)} & \text{for } p < \frac{1+T(p)m(p)}{1+Cm(p)} \\ \frac{1}{1+Cm(p)} & \text{for } p \geq \frac{1+T(p)m(p)}{1+Cm(p)} \end{cases} \quad (1)$$

When the number of threads is small, complete overlapping of network latency is not possible. Processor utilization with one thread is $1/(1 + m(1)T(1))$. Ideally, with p threads available to overlap network delays, the utilization would increase $p-$fold. In practice, because the miss rate and network latency increase to $m(p)$ and $T(p)$, the utilization becomes $p/(1 + m(p)T(p))$.

When it is possible to completely overlap network latency, the processor utilization is limited only by the context switching overhead paid on every miss (assuming a context switch happens on a cache miss), and is given by $1/(1 + m(p)C)$.

The cache and network terms $m(p)$ and $T(p)$ are derived in [2]. These models have been validated through simulations. Both these terms are shown to be the sum of two components: One component independent of the number of threads p and the other linearly related to p (to first order). Multithreading is shown to be useful when p is small enough that the fixed components dominate.

Let us look at some results for the default set of system parameters given in Table 4. We use a direct network [24] characterized by its dimension n and radix k. The analysis assumes 8000 processors arranged in a three dimensional array. The fixed miss rate comprises first-time fetches of blocks into the cache, and the interference due to multiprocessor coherence invalidations.

Parameter	Value
Memory latency	10 cycles
Network dimension n	3
Network radix k	20
Fixed miss rate	2%
Average packet size	4
Cache block size	16 bytes
Thread working set size	250 blocks
Cache size	64K bytes

Table 4: Default system parameters.

Figure 5 displays the processor utilization as a function of the number of threads resident on the processor. Context switching overhead is 4. The degree to which the cache, network, and overhead components impact overall processor utilization is also shown. The ideal curve shows the increase in processor utilization when both the cache miss rate and network contention correspond to that of a single process, and do not increase with the degree of multithreading p.

We see that as few as four processes yield over 85% utilization for a 4-cycle context-switching overhead, which corresponds to the original APRIL processor. This result is similar to that reported by Weber and Gupta [30] for coarse-grain multithreaded processors. The chief reason a low degree of multithreading is sufficient is that context switches are forced only on cache misses, which are expected to happen infrequently. The marginal benefits of additional processes is seen

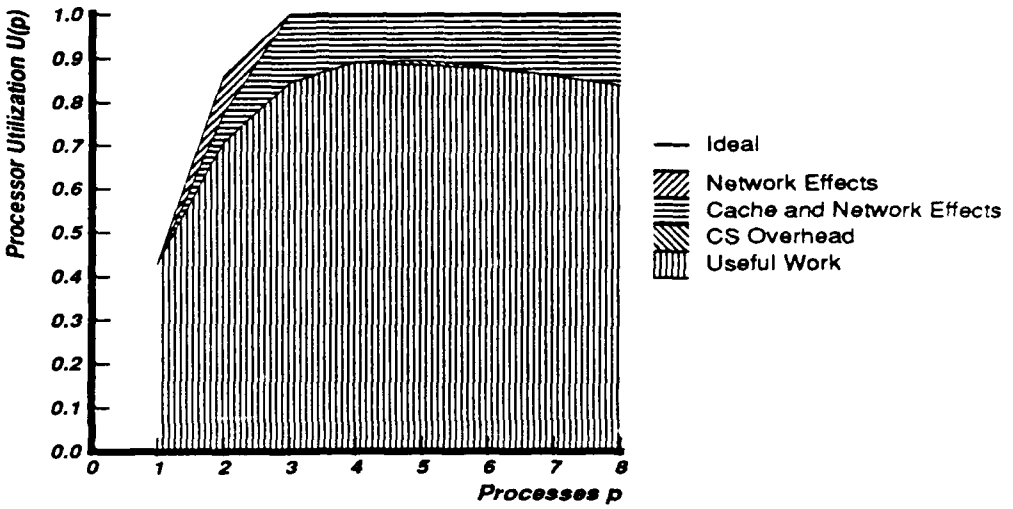

Figure 5: Relative sizes of the cache, network, and overhead components that affect processor utilization. Context switch overhead 4 cycles.

to decrease due to network and cache interference. For example, when the number of processes is increased from one to two, U increases from 0.4 to 0.7, but the corresponding improvement in going from 2 to 3 is just 0.1.

Why is utilization limited to a maximum of about 0.85 despite having an ample supply of threads? The reason is that *available network bandwidth limits the maximum rate at which computation can proceed.* When available network bandwidth is used up, adding more processes will not improve processor utilization. On the contrary, more processes will degrade performance due to the increased cache interference. In such a situation, for better system performance, effort is best spent in increasing the network bandwidth, or in reducing the bandwidth requirement of each thread.

The four-cycle context switch overhead does not significantly impact performance for the default set of parameters. Recall that context-switch overhead plays a role only when the number of threads (plus overhead) is large enough to completely overlap network latency. Because the network bandwidth saturates at 4 processes, network latency increases in proportion to the number of processes, and it is not possible to completely overlap network delay even with an infinite supply of threads.

Figure 6 displays the processor utilization for an 10 cycle context-switching overhead, which corresponds to our initial SPARC-based APRIL design. The limiting utilization is still remarkably high. Because utilization depends on the product of context switching frequency and switching overhead, and because the switching frequency is expected to be small in a cache-based system, the relatively large overhead of 10 cycles can be tolerated. This observation is important because it allows a simpler processor implementation, and is exploited in the design of APRIL.

A multithreaded processor requires larger caches to sustain the working sets of multiple

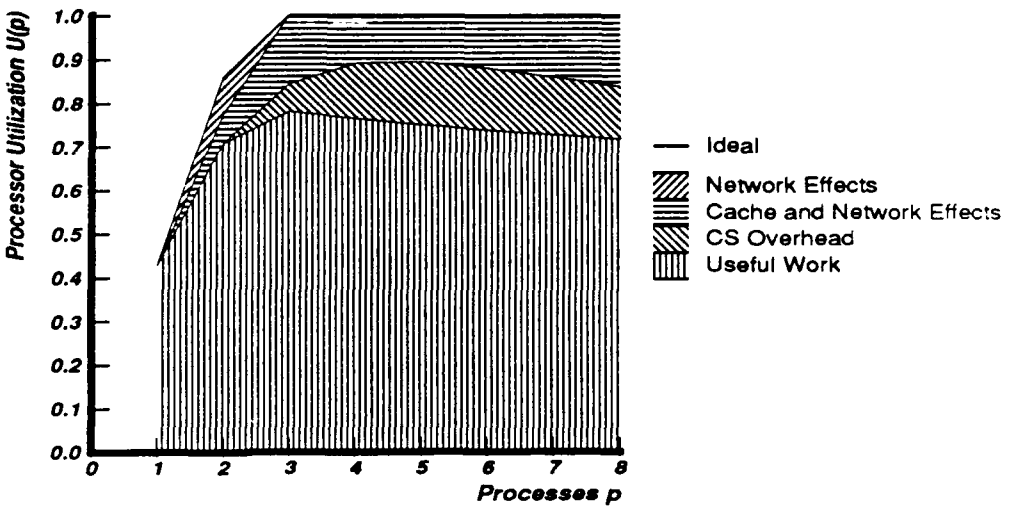

Figure 6: Relative sizes of the cache, network, and overhead components that affect processor utilization. Context switch overhead 10 cycles.

processes, although cache interference is mitigated if the processes share code and data in the cache. For the default parameter set, we found that caches greater than 64K bytes comfortably sustain the working sets of four processes. Smaller caches suffer more interference and reduce the benefits of multithreading. For example, two processes achieved 80% processor utilization in a 256K byte cache, while a comparable utilization required three processes in a 64K byte cache.

10 Conclusions

We described the architecture of APRIL – a coarse-grain multithreaded processor to be used in a cache-coherent multiprocessor called ALEWIFE. By rapidly switching to an alternate task, APRIL can hide communication and synchronization delays and achieve a high processor utilization. Because the processor is rarely idle, it makes effective use of the available network bandwidth. APRIL provides support for fine-grain tasking and detection of futures. APRIL achieves high single-thread performance by executing instructions from a given task until an exception condition like a synchronization fault or remote memory operation occurs and achieves high single-thread performance. Coherent caches reduce the context switch ratio to approximately once every 50-100 cycles. Therefore context switch overheads in the 4-10 cycle range are tolerable, significantly simplifying the processor design. Furthermore, by providing hardware support only for performance-critical operations and migrating other functionality into the compiler and the runtime system, we were able to simplify the processor design even further.

We described a SPARC-based implementation of APRIL that uses the register windows of SPARC as task frames for multiple threads. A processor simulator and an APRIL compiler and runtime system have been written. The SPARC-based implementation of APRIL switches contexts in 11 cycles. APRIL and its associated runtime system practically eliminate the overhead

of fine-grain task creation and detection of futures. For example, for the futures-based Mul-T language, the overhead reduces from 100% on an Encore Multimax-based implementation to under 5% on APRIL. We evaluated the scalability of multithreaded processors in large-scale parallel machines using an analytical model. For typical system parameters and a 10 cycle context-switching overhead, the processor can achieve close to 80% utilization with 3 processor resident threads.

11 Acknowledgements

We would like to acknowledge the contributions of the members the ALEWIFE research group. In particular, Dan Nussbaum was partly responsible for the processor simulator and run-time system and was the source of a gamut of ideas, David Chaiken wrote the cache simulator, Kirk Johnson supplied the benchmarks, and Gino Maa and Sue Lee wrote the network simulator. We appreciate help from Gene Hill, Mark Perry, and Jim Pena from LSI Logic Corporation for the SPARC-based implementation effort. Our design was influenced by Bert Halstead's work on multithreaded processors. Our research benefited significantly from discussions with Bert Halstead, Tom Knight, Greg Papadopoulos, Juan Loaiza, Bill Dally, Steve Ward, Rishiyur Nikhil, Arvind, and John Hennessy. Beng-Hong Lim is partly supported by an Analog Devices Fellowship. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, by grants from the Sloan foundation and IBM.

References

[1] *SPARC Architecture Manual.* 1988. SUN Microsystems, Mountain View, California.

[2] Anant Agarwal. Performance tradeoffs in multithreaded processors. September 1989. Laboratory for Computer Science, Massachusetts Institute of Technololgy.

[3] Arvind and Robert A. Iannucci. *Two Fundamental Issues in Multiprocessing.* Technical Report TM 330, MIT, Laboratory for Computer Science, October 1987.

[4] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, September/October 1986.

[5] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9–24, August 1988.

[6] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Scalability of Directory-Based Cache-Coherent Multiprocessors. November 1989. Laboratory for Computer Science, Massachusetts Institute of Technology.

[7] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 9–21, February 1988.

[8] M. D. Hill et al. Design Decisions in SPUR. *Computer*, 19(10):8–22, November 1986.

[9] Mark Horowitz et al. A 32-Bit Microprocessor with 2K-Byte On-Chip Instruction Cache. *IEEE Journal of Solid-State Circuits*, October 1987.

[10] W. J. Dally et. al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, IEEE, New York, June 1987.

[11] R.H. Halstead and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, IEEE, New York, June 1988.

[12] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[13] J. L. Hennessy and T. R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[14] J. L. Hennessy, N. Jouppi, F. Baskett, T. R. Gross, and J. Gill. Hardware/Software Tradeoffs for Increased Performance. In *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems*, pages 2–11, March 1982. ACM, Palo Alto, CA.

[15] R.A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Hawaii, June 1988.

[16] W. J. Kaminsky and E. S. Davidson. Developing a Multiple-Instruction-Stream Single-Chip Processor. *Computer*, 66–78, December 1979.

[17] M. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* Ph.D. Thesis, Computer Science Division (EECS) UCB/CSD 83/141, University of California at Berkeley, October 1983.

[18] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implemenation*, June 1989.

[19] David Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, February 1988. Technical Report YALEU/DCS/RR-632.

[20] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel tasks. November 1989. Submitted for Publication.

[21] Rishiyur S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[22] J. Rees and N. Adams. T: A Dialect of LISP. In *Proceedings of Symposium on Lisp and Functional Programming*, August 1982.

[23] J. Rees and W. Clinger eds. Revised^3 Report on The Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[24] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

[25] Charles L. Seitz. The Cosmic Cube. *CACM*, 28(1):22–33, January 1985.

[26] Dennis Shasha and Marc Snir. *Efficient and Correct Execution of Parallel Programs that Share Memory.* Research Report 58037, Courant Institute, New York University and IBM T.J. Watson Research Center, T.J. Watson Research Center, POB 218, Yorktown Heights, NY 10598, July 1987.

[27] B.J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[28] P. A. Steenkiste and J. L. Hennessy. A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[29] David W. Wall. Global Register Allocation at Link Time. In *SIGPLAN '86. Conference on Compiler Construction*, June 1986.

[30] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[31] Colin Whitby-Strevens. The Transputer. In *Proceedings 12th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1985.

On-line Algorithms for Path Selection in Nonblocking Networks

Sanjeev Arora[1,2]
Tom Leighton[1,2]
Bruce Maggs[2]

[1]Mathematics Department and
[2]Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Nonblocking networks arise in a variety of applications involving communications. The most well known examples include telephone networks, data networks, and distributed memory architectures. Although asymptotically optimal constructions are known for nonblocking networks in a variety of models, it is generally not known how to select paths for the desired network connections efficiently on-line. In this paper, we present the first optimal-time algorithms for path selection in an optimal-size nonblocking network. In particular, we describe a bounded-degree, $O(N \log N)$-switch nonblocking network that can realize any sequence of connections and disconnections among N terminals with $O(\log N)$ bit-step delay. Viewed in the context of a telephone switching network, our network and algorithm can handle any sequence of calls among N parties with $O(\log N)$ bit-step delay per call (even if many calls are made at once). Parties can hang up and call again whenever they like, and multiparty calls can be made without affecting the performance of the algorithm — every call is still put through in $O(\log N)$ time. Viewed in the context of distributed memories for parallel machines, our algorithm allows any processor to access any idle block of memory within $O(\log N)$ bit-steps at any time — no matter what other connections have been made previously or are being made simultaneously.

1 Introduction

Nonblocking networks arise in a variety of communications applications. Common examples include telephone networks, data networks, and network architectures for parallel machines. In a typical application, there are $2N$ *terminals* (usually thought of as N *inputs* and N *outputs*) interconnected by switches (also called nodes) that can be set so as to link the inputs to the outputs with node-disjoint paths according to a specified permutation. The goal is to interconnect the terminals and switches so that any unused input–output pair can be connected by a path of unused switches, no matter what other paths exist at the

This research was supported by the Defense Advanced Research Projects Agency under Contracts N00014–87–K–825 and N00014–89–J–1988, the Air Force under Contract AFOSR–89–0271, and the Army under Contract DAAL–03–86–K–0171.

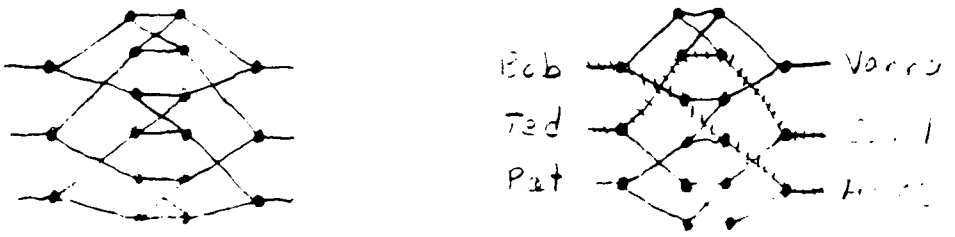

Figure 1: A nonblocking network with 3 inputs and 3 outputs.

time. Such a network is said to be *nonblocking*. For example, the 6-terminal graph shown in Figure 1 is nonblocking since no matter which input–output pairs are connected by a path, there is a node-disjoint path linking any unused input–output pair. In particular, if Bob is talking to Alice and Ted is talking to Carol, then Pat can still call Vanna.

Substantial progress has been made on the problem of building nonblocking networks with small numbers of switches. Shannon [15] proved that any $2N$-terminal nonblocking network must have $\Omega(N \log N)$ switches. Clos [5] and, later, others [6, 7, 14] discovered nonblocking networks with $O(N^{1+\epsilon})$ switches and/or wires. Pippenger [13] discovered an $O(N \log^2 N)$-switch nonblocking network. Benes [4] described networks with $O(N \log N)$ switches capable of realizing any 1–1 connection of inputs to outputs with node-disjoint paths provided that all the connections to be made are known in advance, and provided that once made, a connection cannot be broken. Such networks are called *rearrangeable*, and are not as powerful as nonblocking networks. The existence of an $O(N \log N)$-switch nonblocking network was first proved by Bassalygo and Pinsker [3]. Although the Bassalygo and Pinsker proof is nonconstructive, subsequent work on the explicit construction of expanders [11] yielded a construction for an $O(N \log N)$-switch nonblocking network. This result was later extended by Feldman, Friedman, and Pippenger [7] who discovered constructions of nonblocking networks which had the additional property that each input can be simultaneously connected to an arbitrary set of outputs provided that every output is connected to just one input. Such networks are called *generalized nonblocking networks*, and are useful in the context of multiparty calling in a telephone network as well as for broadcasting in a parallel machine. Recently Turner [16] found constructions of such networks that are somewhat simpler but slightly larger than the networks in [7].

Unfortunately, there has not been as much progress on the problem of setting the switches so as to realize the connection paths. Indeed, many of the references cited previously show that there exists a way of setting the switches so as to realize the desired paths, but are unable to provide any reasonable algorithms (on-line or off-line) for actually finding the right switch settings. There are some exceptions. For example, it is possible to find paths on-line in $O(\log N)$ time for the naive $\Theta(N^2)$-switch nonblocking networks (e.g. an $N \times N$ mesh of trees [8]), and in polylogarithmic time for some of the $\Theta(N^{1+\epsilon})$-switch constructions mentioned previously. More recently, Lin [12] found polylogarithmic time path selection algorithms for $O(N \log^2 N)$-switch networks. No fast algorithms were known for the $O(N \log N)$-switch networks, however, and no $O(\log N)$-step algorithms were known for any of the $o(N^{1+\epsilon})$-switch networks.

In this paper, we describe an $O(N \log N)$-switch nonblocking network for which each path connection can be made on-line in $O(\log N)$ bit-steps. Moreover, the network can realize any multiparty call in $O(\log N)$ word-steps on-line, provided that all the parties to each call are known at the time of the call. (If all of the parties are not known in advance, then there is a cost of $O(\log N)$ steps for each group of callers added later on.)

The algorithms work even if many calls are made at once — every call still gets through in $O(\log N)$ bit-steps, no matter what calls were made previously and no matter what calls are currently active, provided that no two inputs try to access the same output at the same time. (If many inputs inadvertently try to access the same output at the same time, all but one of the inputs will receive a busy signal. The busy signals are also returned in $O(\log N)$ bit-steps, but, at present, we require the use of the AKS sorting circuit [2] to generate the busy signals. Alternatively, we could merge the calling parties together, but this also requires the use of the AKS sorting circuit.) In all scenarios, the size of the network and the speed of the path selection algorithm are asymptotically optimal.

In addition to providing the first optimal solution to the abstract telephone switching problem, our results significantly improve upon previously known algorithms for bit-serial (or byte-serial) packet routing. Previously, $O(\log N)$-bit-step algorithms for packet routing were known only in the special case where are all packet paths are created or destroyed at the same time, and even then only by resorting to the AKS network, or by using randomness on the hypercube [1]. In many circuit-switched parallel machines, however, packets are of varying lengths and packet paths are created and destroyed at arbitrary times, thereby requiring that paths be routed in a nonblocking fashion – which is something that previously discovered algorithms were not capable of doing. Even without worrying about the nonblocking property, our results provide the first non-AKS $O(\log N)$-bit-step algorithms for bit-serial packet routing on a bounded-degree network. Although we have not yet tested our algorithms experimentally, we are optimistic that they will perform well in practice.

The family of networks that we use to obtain these results combines expanders and the Benes network in a manner similar to the multibutterfly network described by Upfal [17]. We refer to the networks as *multi-Benes networks*. The details of the construction are provided in Section 2 of the paper. We can also apply our results to bandwidth-limited switching networks such as fat-trees [10], and obtain optimal performance in terms of load factor. Such results may be more useful in the context of telephone networks, where there are limitations on the number of calls based on the proximity of the calls (e.g., it is not possible for everyone on the East Coast to call everyone on the West Coast at the same time).

The description and analysis of the path selection algorithms is divided into three sections. In Section 3, we describe on-line algorithms for adding a single connection path in the network. These algorithms are similar to the fault-tolerant routing algorithms in [9]. Indeed, we can think of currently-used wires as being faulty since they cannot be used to form a new connection path. Similarly, the algorithms we describe for routing in nonblocking networks can easily be extended to be highly tolerant to faults in the network.

In Section 4, we describe an $O(\log N)$-bit-step algorithm for bit-serial routing in a multibutterfly. This algorithm relies on a unique-neighbor property possessed by all highly-expanding graphs. By implementing this algorithm on the multi-Benes network and combining it with the methods of Section 3, we produce an algorithm that can handle many calls at the same time, independent of what calls have been made previously and what calls are currently connected.

In Section 5, we describe algorithms for handling multiparty calls, and situations where many inputs try to reach the same output simultaneously. Some of these algorithms rely on the AKS sorting circuit and are not as practical as those described in Sections 3 and 4.

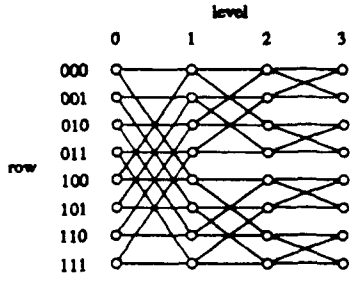

Figure 2: An 8-input butterfly network.

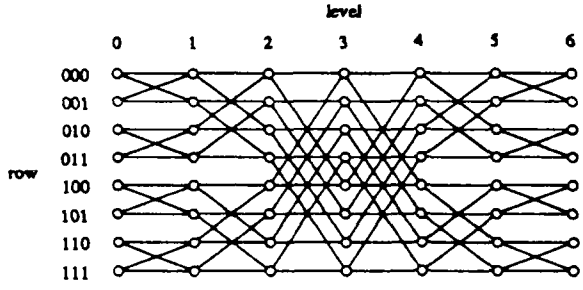

Figure 3: An 8-input Benes network.

2 The multi-Benes and multibutterfly networks

Our nonblocking network is constructed from a Benes network and expander graphs in much the same way as a multibutterfly network is constructed from a butterfly network. We start by describing the butterfly, Benes, and multibutterfly networks.

An N-input butterfly has $\lg N + 1$ levels, each with N-nodes. An example is shown in Figure 2. The Benes network is a $(2\lg N + 1)$-level network consisting of back-to-back butterflies. For example, see Figure 3.

A multibutterfly is formed by gluing together butterflies in a somewhat unusual way. In particular, given 2 N-input butterflies G_1 and G_2 and a collection of permutations $\Pi = \langle \pi_0, \pi_1, \ldots, \pi_{\lg N} \rangle$ where $\pi_l : [0, \frac{N}{2^l} - 1] \to [0, \frac{N}{2^l} - 1]$, a 2-butterfly is formed by merging the node in row $\frac{iN}{2^l} + i$ of level l of G_1 with the node in row $\frac{iN}{2^l} + \pi_l(i)$ of level l of G_2 for all $0 \le i \le \frac{N}{2^l} - 1$, all $0 \le j \le 2^l - 1$, and all $0 \le l \le \lg N$. The result is an N-input $(\lg N + 1)$-level graph in which each node has 4 inputs and 4 outputs. Of the 4 output edges at a node, two are *up* outputs and two are *down* outputs (with one up edge and one down edge coming from each butterfly). Multibutterflies (i.e., d-butterflies) are composed from d butterflies in a similar fashion using $d - 1$ sets of permutations, $\Pi^{(1)}, \ldots, \Pi^{(d-1)}$, resulting in a $(\lg N + 1)$ level network with $2d \times 2d$ switches. For example, see Figure 4.

The notion of up and down edges can be formalized in terms of splitters. More precisely, the edges from level l to level $l + 1$ in rows $\frac{iN}{2^l}$ to $\frac{(j+1)N}{2^l} - 1$ in a multibutterfly form a *splitter* for all $0 \le l < \lg N$ and $0 \le j \le 2^l - 1$. Each of the 2^l splitters starting at level l has $\frac{N}{2^l}$ inputs and outputs. The outputs on level $l + 1$ are naturally divided into $\frac{N}{2^{(l+1)}}$ *up*

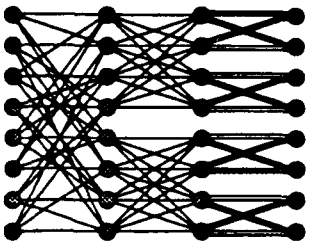

Figure 4: An 8-input 2-butterfly network.

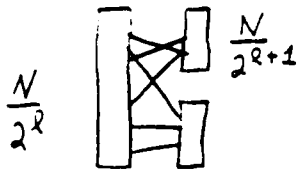

Figure 5: A splitter.

outputs and $\frac{N}{2^{(l+1)}}$ *down* outputs. For example, see Figure 5. By definition, all splitters on the same level l are isomorphic, and each input is connected to d up outputs and d down outputs according to the butterfly and the permutations $\pi_l^{(1)}, \ldots, \pi_l^{(d-1)}$. Hence, any input and output of the multibutterfly are connected by a single logical (up-down) path through the multibutterfly, but each step of the logical path can be taken on any one of d edges. For example, see Figure 6.

The most important characteristic of a multibutterfly is the set of permutations $\Pi^{(1)}$, ..., $\Pi^{(d-1)}$ that prescribe the way in which the component butterflies are to be merged. For example, if all of the permutations are the identity map, then the result is the *dilated butterfly* (i.e., a butterfly with d copies of each edge). We are most interested in multibutterflies that have expansion properties. In particular, we say that an M-input splitter has *expansion property* (α, β) if every set of $k \leq \alpha M$ inputs is connected to at least βk up outputs and βk down outputs for $\beta > 1$. Similarly, we say that a multibutterfly has *expansion property* (α, β) if each of its component splitters has expansion property (α, β). For example, see Figure 7.

If the permutations $\Pi^{(1)}, \ldots, \Pi^{(d-1)}$ are chosen randomly, then with good probability the resulting d-butterfly has expansion property (α, β) for any d, α, and β for which $2\alpha\beta < 1$ and

$$d < \beta + 1 + \frac{\beta + 1 + \ln 2\beta}{\ln(\frac{1}{2\alpha\beta})}. \quad (1)$$

Constructions for splitters and multibutterflies with good expansion properties are known although the expansion properties are generally not as good as those obtained from randomly-generated graphs.

Like a multibutterfly, a multi-Benes network is formed from Benes networks by merging them together. A 2-multi-Benes network is shown in Figure 8. An N-input multi-Benes network has $2\lg N + 1$ levels labeled 0 through $2\lg N$. Levels $\lg N$ through $2\lg N$ form a

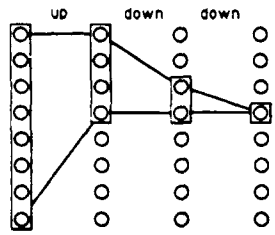

Figure 6: The logical path from an input to output 011.

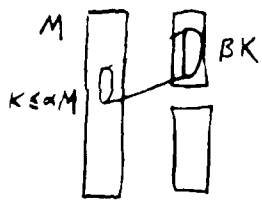

Figure 7: A splitter with expansion property (α, β).

multibutterfly, while levels 0 through $\lg N$ form the mirror image of a multibutterfly.

As in the multibutterfly, the edges in levels $\lg N$ through $2\lg N$ are partitioned into splitters. Between levels 0 and $\lg N$, however, the edges are partitioned into *mergers*. More precisely, the edges from level l to level $l+1$ in rows $j2^{l+1}$ to $(j+1)2^{l+1}-1$ form a *merger* for all $0 \leq l < \lg N$ and $0 \leq j \leq N/2^{l+1}-1$. For example, see Figure 9. Each of the $N/2^{l+1}$ mergers starting at level l has 2^{l+1} inputs and outputs. The inputs on level l are naturally divided into 2^l *up* inputs and 2^l *down* inputs. All mergers on the same level l are isomorphic, and each input is connected to $2d$ outputs. There is a single (trivial) logical path from any input of a multi-Benes network through the mergers on the first $\lg N$ levels to the single splitter on level $\lg N$. From level $\lg N$ there is a single logical path through the the splitters to any output. In both cases, the logical path can be realized by many physical paths.

We say that an M-output merger has expansion property (α, β) if every set of $k \leq \alpha M$ inputs (up or down) is connected to at least $2\beta k$ outputs $\beta > 1$. With nonzero probability, a random set of permutations yields a merger with expansion property (α, β) for any d, α, and β for which $\alpha\beta < 1$ and

$$2d \;<\; 2\beta + 1 + \frac{2\beta + 1 + \ln 2\beta}{\ln(\frac{1}{2\alpha\beta})}. \tag{2}$$

We say that a multi-Benes network has expansion property (α, β) if each of its component mergers and splitters has expansion property (α, β). The multibutterflies and multi-Benes networks considered throughout this paper are assumed to have expansion property (α, β).

3 A non-blocking path selection algorithm for the multi-Benes network

In this section we describe an efficient on-line algorithm for satisfying connection requests in a multi-Benes network. The algorithm establishes a path from an unused input to an

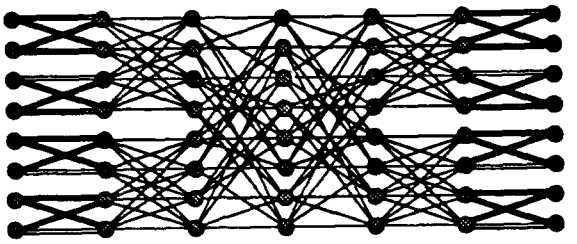

Figure 8: An 8-input 2-multi-Benes network.

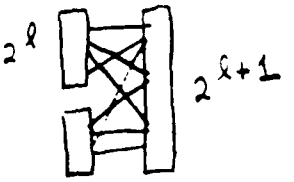

Figure 9: A merger.

unused output in $O(\log N)$ bit-steps, where N is the number of rows. Although non-blocking networks of size $O(N \log N)$ and depth $O(\log N)$ were previously known [3], no efficient algorithms for establishing paths in them were known. Throughout, we assume that at most one input tries to access any output at a time, and that each input accesses at most one output at a time. Algorithms for multiparty calling are deferred to Section 5.

In order for the algorithm to succeed, the multi-Benes network must be lightly loaded by some fixed constant factor L. Thus, in an N-row multi-Benes network, we only make connections between the N/L inputs and outputs in rows that are multiples of L. Since the other inputs and outputs are not used, the first and last $\lg L$ levels of the network can be removed, and the N/L inputs and outputs can each be connected directly to their L descendants and ancestors on levels $\lg L$ and $2\lg N - \lg L$, respectively.

The basic idea is to treat the switches through which paths have already been established as if they were faulty and to apply the fault propagation techniques from [9] to the network. In particular, we define a node to be *busy* if there is a path currently routing through it, and we recursively define a node to be *blocked* if all of its up outputs or all of its down outputs are busy or blocked. More precisely, switches are declared to be blocked according to the following rule. Working backwards from level $2\lg N - \lg L - 1$ to level $\lg N$, a switch is declared blocked if either all d of its up edges or all d of its down edges lead to busy or blocked switches. From level $\lg N - 1$ to level $\lg L$, a switch is declared blocked if all $2d$ of its outputs lead to busy or blocked switches. A switch that is neither busy nor blocked is said to be *working*.

The following pair of lemmas bound the fraction of input switches that are blocked in

every splitter and merger.

Lemma 1 *For $L > 1/2\alpha(\beta - 1)$, at most a 2α fraction of the inputs in any splitter are declared to be blocked. Furthermore, at most an α fraction of the switches are blocked because of busy and blocked switches from the upper outputs, and at most an α fraction are blocked because of the lower outputs.*

Proof: The proof is by induction on level number, starting at level $2\lg N - \lg L$ and working backwards to level $\lg N$. The base case is trivial since there are no blocked switches on level $2\lg N - \lg L$. Suppose the inputs of an M-input splitter contain more than αM switches that are blocked because of the upper (say) outputs. Consider the set U of busy or blocked upper outputs. Since all of the edges out of a blocked input lead to busy or blocked outputs, we can conclude that $|U| \geq \alpha\beta M$. Since all paths passing through the upper outputs must lead to one of $M/2L$ terminals, there can be at most $M/2L$ busy switches among the upper outputs of the splitter. Furthermore, by induction there are at most αM blocked switches among the upper outputs. Thus, $|U| \leq \alpha M + M/2L$. For $L > 1/2\alpha(\beta - 1)$ we have a contradiction. Hence, at most an α fraction of the switches are blocked or busy, as claimed. □

Lemma 2 *For $L > 1/2\alpha(\beta - 1)$, at most a 2α fraction of the upper inputs and a 2α fraction of the lower inputs in any merger are blocked.*

Proof: The proof is like that of Lemma 1 □

After the fault propagation process, every working switch in the first half of the network has an output that leads to a working switch, and every working switch in the second half has both an up output and a down output that lead to working switches. Furthermore, since at most a 2α fraction of the switches in each merger on level $\lg L$ are blocked, each of the N/L inputs has an edge to a working switch on level L. At the other end, each of the N/L outputs can be reached by a working switch on level $2\lg N - \lg L$. As a consequence, we can establish a path through working switches from any unused input to any unused output in $O(\log N)$ bit-steps using a simple greedy algorithm. Since the declaration of blocked switches takes just $O(\log N)$ bit-steps, and since the greedy routing algorithm is easily accomplished in $O(\log N)$ bit-steps, the entire process takes just $O(\log N)$ bit-steps.

4 Establishing many paths at once

The preceding algorithm can establish any single additional path successfully in $O(\log N)$ bit-steps. While this is sufficient to show that the multi-Benes network is nonblocking, it is not sufficient to handle many calls at once. In what follows, we describe an algorithm for routing an arbitrary number of additional calls in $O(\log N)$ bit-steps. As before, we assume for the time being that each input and output is involved in a at most one two-party call. Extensions to the algorithm for handling multiparty calls are described in Section 5. We also assume that paths are established between inputs and outputs on rows congruent to 0 mod L in the multi-Benes network, where $L \geq 1/\alpha$. This will insure that no splitter or merger is ever overloaded.

To simplify the exposition of the algorithm, we start by describing an algorithm for routing any initial set of paths in a multibutterfly (i.e., we don't worry about the nonblocking aspect of the problem for the time being). This comprises the first known circuit-switching algorithm for the multibutterfly. (Previous routing algorithms for the multibutterfly [9, 17]

only worked for the store-and-forward model of routing.) We then modify the blocking definition of Section 3 and show how to implement the algorithm in a nonblocking fashion on the multi-Benes network.

4.1 Circuit-switching on a multibutterfly

Our circuit-switching algorithm requires the splitters in the multibutterfly to have a special "unique-neighbors" property defined as follows.

Definition 3 *An M-input splitter is said to have the (α, δ) unique neighbor property if in every subset X of $k \leq \alpha M$ inputs, there are δk nodes in X which have an up-output neighbor that is not adjacent to any other node in X, and there are δk nodes in X which have a down-output neighbor that is not adjacent to any other node in X (i.e., δk nodes in X have a unique up-neighbor, and δk nodes have a unique down-neighbor).*

Lemma 4 *Any splitter with the (α, β) expansion property has the (α, δ) unique-neighbors property where $\delta = 2\beta/d - 1$, provided that $\beta > d/2$.*

Proof: Consider any set X of $k \leq \alpha M$ inputs in an M-input splitter. These nodes have βk neighbors among the up (down) outputs. Let n_1 denote the number of these neighbors incident to precisely one node of X, and let n_2 denote the number of neighbors incident to two or more nodes of X. Then $n_1 + n_2 \geq \beta k$ and $n_1 + 2n_2 \leq dk$. Solving for n_1 reveals that $n_1 \geq (2\beta - d)k$. Hence at least $(2\beta/d - 1)k$ of the nodes in X are adjacent to a unique neighbor. □

By Equation 1, we know that randomly generated splitters have the (α, δ) unique-neighbors property where δ approaches 1 as d gets large and α gets small. Explicit constructions of such splitters are not known, however. Nevertheless, we will consider only multibutterflies with the (α, δ) unique-neighbors property for $\delta > 0$ in what follows.

Remark: The (α, β) expansion property ($\beta > d/2$) is a sufficient condition for the unique-neighbors property, but by no means necessary. In fact, we can easily prove the existence of random splitters which have a fairly strong (α, δ) unique-neighbors property for small degree. For such graphs, the routing algorithm we are about to describe is more efficient in terms of hardware required. However, multibutterflies with expansion properties will remain the object of our focus.

It is relatively easy to extend paths from one level to the next in a multibutterfly with the (α, δ) unique-neighbors property. The reason is that those paths at switches with unique neighbors can be trivially extended without worrying about blocking any other path trying to reach the next level. By proceeding recursively, it is easy to see that all the paths can be extended from level l to level $l + 1$ (for any l) in $\log(N/L2^l)/\log(1/1 - \delta)$ steps. In particular, a "step" consists of:

1. every path still waiting to be extended sends out a "proposal" to his output (level $l + 1$) neighbors in the desired direction (up or down),
2. every output node that receives precisely one proposal sends back its acceptance to that proposal,
3. every path receiving an acceptance advances to one of its accepting outputs on level $l + 1$.

Splitters connecting level l to level $l+1$ have $M = N/2^l$ inputs and at most M/L paths can pass through them by definition of L. Since $L > 1/\alpha$, the set of switches containing paths needing to be extended has size at most αM and we can apply the (α, δ) unique-neighbors property to ensure that at each step, the number of paths still remaining to be extended decreases by a $(1-\delta)$ factor. Hence, all of the paths are extended in $\log(N/L2^l)/\log(1/(1-\delta))$ steps, as claimed.

By using the path extension algorithm just described on each level in sequence, we can construct all of the paths in

$$\sum_{l=0}^{\lg N-1} \frac{\log \frac{N}{L2^l}}{\log \frac{1}{1-\delta}} \leq \frac{\log^2 \frac{N}{L2^l}}{2\log \frac{1}{1-\delta}} = O(\log^2 N)$$

bit-steps. To construct the paths in $O(\log N)$ bit-steps we modify this algorithm as follows:

> Given a fraction $< \alpha$ of paths that need to be extended at an M-input splitter, the algorithm does not wait $O(\log M)$ time for every path to be extended before it begins the extension at the next level. Instead, it waits only $O(1)$ steps, in which time the number of unextended paths falls to a fraction ρ of its original value, where $\rho < 1/d$. Now the path extension process can start at the next level. The danger here is that the ρ fraction of paths left behind may find themselves blocked by the time they reach the next level, and so we need to ensure that this won't happen. Therefore, stalled paths send out *place-holders* to all of their neighbors at the next level, and henceforth the neighbors with place-holders participate in path extension at the next level, as if they were paths. Of course, the neighbors holding place-holders must in general extend in both the upper and the lower output portions of the splitter, since they don't know yet which path will ultimately use them.

Notice that a place-holder not only reserves a spot that may be used by a path at a future time, but also helps to chart out the path by continuing to extend ahead. In order to prevent place-holders from multiplying too rapidly and clogging the system — since if the fraction of inputs of a splitter which are trying to extend rises above α, the path extension algorithm ceases to work — we need to ensure that as stalled paths get extended, they send *cancellation signals* to the placeholding nodes ahead of them to tell them they are not needed anymore. When a placeholding node gets cancellations from all the nodes who had requested it to hold their place, it disappears and ceases to extend anymore. It also sends cancellations to any nodes ahead of it that may be holding a place for it. As we shall see, this scheme prevents place-holders from getting too numerous.

The $O(\log N)$-step algorithm for routing paths proceeds in *phases* consisting of the following two types of steps:

- C steps of passing cancellation signals. These cancellation signals travel at the rate of one level per step,
- T steps of extending from one level to the next. In this time, the number of stalled (i.e., unextended) paths at each splitter drops by least a factor of ρ, where $\rho \leq (1-\delta)^T$.

Each path is restricted to extend forward by at most one level during each phase. We refer to the first wave of paths and placeholders to arrive at a level as the *wavefront*. The

wavefront moves forward by one level during each phase. If a path or placeholder in the wavefront isn't extended in T steps, it sends placeholders to all of its neighbors at the end of the phase. We will assume that $C \geq 2$ so that cancellation signals have a chance to catch up with the wavefront, and that $d \geq 3$.

The following lemmas will be useful in proving that every path is extended to completion in $\lg N$ phases provided that $L \geq 2/\alpha$ and $\rho < 1/13d$. The key to our approach is to focus on the number of stalled paths (corresponding to real paths *or* placeholders) at the inputs of each splitter. In particular, we let $S(i,t)$ denote the maximum fraction of inputs of any splitter at level i that contain stalled paths at the end of phase t of the algorithm. By definition, $S(i,t) = 0$ for $t < i$, since the wavefront arrives at level i at phase i. Each stalled path generates up to $2d$ placeholders at the next level, which might later become stalled themselves, so it is crucial to keep the number of stalled paths at each level small.

Lemma 5 *If the number of paths (real or placeholder) reaching the inputs of a level i splitter when the wavefront arrives is less than an α fraction of the inputs, then $S(i,t) \leq \rho^{t-i} S(i,i)$ for $t \geq i$.*

Proof: In each phase of the algorithm, the number of stalled paths at the inputs drops by a factor of ρ, provided the number of paths trying to extend is never greater than an α fraction of the inputs of the splitter. Since the number of paths reaching the inputs never increases after the wavefront arrives, this condition is always satisfied. □

Let P_i denote the fraction of inputs containing paths (real and placeholder) in a level i splitter when the wavefront arrives (i.e., at the end of phase $i-1$). Note that if $P_i \leq \alpha$, then $S(i,i) \leq \rho P_i$.

Lemma 6

$$P_i \leq \frac{1}{L} + 2dS(i-1,i-1) + \sum_{k=2}^{C} 2^k dS(i-k,i-2) + \sum_{l=1}^{\infty}\sum_{k=1}^{C} 2^{Cl+k} dS(i-Cl-k, i-l-2).$$

Proof: Note that $1/L$ upperbounds the fraction of real paths that could be in the wavefront, since that is the number of real paths that will ever pass through the splitter. The $2dS(i-1,i-1)$ term represents the fraction of inputs that could have placeholders generated by stalled paths at level $i-1$ (the factor 2 comes in because the number of inputs in a splitter at level $i-1$ is twice as many as those in a level i splitter). Next, $4dS(i-2,i-2)$ upperbounds the fraction of inputs containing placeholders generated by paths stalled at level $i-2$, but whose placeholders were extended from level $i-1$ to level i. (Note that if $C \geq 3$, for example, we have an $8dS(i-3,i-2)$ contribution instead of an $8dS(i-3,i-3)$ contribution to P_i since paths stalled in level $i-3$ during phase $i-3$, but getting through during phase $i-2$, send a cancellation signal to levels $i-2$, $i-1$, and i during phase $i-1$, and hence they do not contribute placeholders to level i during phase i.) The rest of the terms in the summation may be counted similarly. Finally, though our summation seems to have infinitely many terms, only finitely many of them are non-zero. □

Lemma 7 *The fraction of inputs containing paths (real and placeholders) at any splitter is never more than α provided $L \geq \frac{2}{\alpha}$, $\rho \leq \frac{1}{13d}$, $d \geq 3$, and $C \geq 3$.*

Proof: We prove by induction on i that for $\gamma = \frac{\alpha}{13d}$, $S(i,i) < \gamma$ and $P_i < \alpha$. We only need to prove the inductive step, since the base case is trivial. By the recurrence of Lemma 6, we have:

$$
\begin{aligned}
P_i &\leq \frac{1}{L} + 2d\gamma + \sum_{k=2}^{C} 2^k d\gamma\rho^{k-2} + \sum_{l=1}^{\infty}\sum_{k=1}^{C} 2^{Cl+k} d\gamma\rho^{(C-1)l+k-2} \\
&\leq \frac{1}{L} + 2d\gamma + \frac{4d\gamma(1-(2\rho)^{C-1})}{1-2\rho} + \frac{d\gamma 2^{C+1}\rho^{C-2}(1-(2\rho)^C)}{(1-2^C\rho^{C-1})(1-2\rho)} \\
&\leq \frac{1}{L} + 2d\gamma + 4.06d\gamma + .3d\gamma.
\end{aligned}
$$

Note that we have used the fact that $d \geq 3$, $C \geq 3$, and $\rho \leq 1/13d$. (We really only needed $C \geq 2$, but the constants are better for $C \geq 3$.) Thus if $\gamma = \alpha/13d$ and $L > 2/\alpha$, then $P_i \leq \alpha$. Also, as we noted earlier, $S(i,i) \leq \rho P_i$ and if $\rho < 1/13d$, by Lemma 6, we have: $S(i,i) < \alpha/13d = \gamma$, thereby establishing the induction. □

From Lemma 7, it is clear that no splitter ever has more than an α fraction of its inputs containing paths to be extended to outputs. Therefore the path extension algorithm never is swamped by placeholders and always works as planned at each level, cutting down the number of stalled paths by a factor of ρ during each phase. Hence, $\lg(M/L)/\lg(1/\rho)$ phases after the wavefront arrives at a splitter of size M, all paths are extended. Thus, the algorithm establishes all paths successfully in

$$
\begin{aligned}
\min_{0\leq i\leq \lg(N/L)} \left(i + \frac{\log\frac{N}{2^i L}}{\log\frac{1}{\rho}} + \frac{\log\frac{N}{L} - i}{C} \right) &= \min_{0\leq i\leq \lg(N/L)} \left(\frac{\log\frac{N}{L}}{\log\frac{1}{\rho}} + \frac{\log\frac{N}{L}}{C} + i\left(1 - \frac{1}{\log\frac{1}{\rho}} - \frac{1}{C}\right)\right) \\
&= \log(N/L)
\end{aligned}
$$

phases, provided $\rho < 1/4$. This is because if a path is last stalled at level i, then it passes through level i by phase $i + \log(N/2^i L)/\log(1/\rho)$ and reaches the end $(\log(N/L) - i)/C$ phases later. At first, this result seems too good to be true, but recall that stalled re- paths catch up to the wavefront very quickly once they get through, and that they get through at a very high rate. Hence, all real paths get through to the final level along with the wavefront!

By propagating the cancellations from the previous phase at the same time as the paths are extended, a single phase can be implemented in $\max(C, T+1)$ steps. (The extra step in $T+1$ is for initiating placeholders for stalled paths.) By using very good splitters ($\delta \approx 1$), α small, d large, $C = 3$, and $T = 1$, we can obtain a $(2+\varepsilon)\log N$ step algorithm for routing all the paths. It is worth noting that this beats the best previous bounds for store and forward routing [9] by a factor of 2. Unfortunately, d and L need to be quite large to achieve this bound. For more reasonable values of d (< 10) and L (< 100), we can achieve provable routing times of about $100 \lg N$. Fortunately, the algorithms appear to work faster in reality. We plan to have experimental data demonstrating this point in the final draft of the paper.

It is also worth noting that each switch needs only to keep track of a few bits of information to make its decisions. This is because only the ith bit of the destination is needed to make a switching decision at level i, and therefore a switch at that level looks at this bit, strips it off, and passes the rest of the destination address onward. The path as a whole snakes forward through the network. If it ever gets blocked, the entire snake halts behind it. The implementation details for this scheme are straightforward. Previously, only the AKS sorting circuit was known to achieve this performance for bounded-degree networks, but at a much greater cost in complexity and constant factors.

4.2 Routing many paths in a nonblocking fashion on a multi-Benes network

It is not difficult to implement the algorithm just described on a multi-Benes network. We need to define the unique-neighbor property for mergers, but it is straightforward. If we have to route around existing paths, however, then we have to combine the algorithm with the kind of analysis used in Section 3. In particular, we need to modify the definition of being blocked so that a node on level l is *blocked* if more than $2\beta - d - 1$ of its up (or down) neighbors on level $l+1$ are busy or blocked. (As before, we assume that $\beta > d/2$.) *Working* nodes will then be guaranteed to have at least $2d - 2\beta + 1$ working neighbors. Hence, any set of $k \leq \alpha M$ working inputs in an M-input splitter will have a $(\alpha, 1/d)$ unique-neighbor property, which is sufficient for the routing algorithm to work.

Of course, we must also check that the new blocking definition does not result in any inputs to the multi-Benes network becoming blocked. This can be done with an argument similar to that in Lemma 1. Roughly speaking, if an α fraction of the inputs to an M-input splitter were to become blocked, by upper inputs (say), then each of the inputs is incident to $2\beta - d - 1$ blocked or busy upper inputs. Hence, of the $\alpha\beta M$ upper output neighbors, at most $(2d - 2\beta + 1)\alpha M$ of them can be working. This means that at least a $2(3\beta - 2d - 1)\alpha$ fraction of the upper outputs are busy or blocked. By induction, however, the fraction of the blocked upper outputs is at most α and thus

$$2(3\beta - 2d - 1)\alpha \leq \alpha + \frac{1}{L}$$

which is a contradiction if $L > 1/(6\beta - 4d - 3)$. Of course, for this argument to work, we need d large enough so that it is possible for $\beta > (4d + 3)/6$. It is likely that this can be improved to $\beta > d/2$, but the details will be left to the final paper.

5 Handling multiparty calls

If all the parties of a multiparty call are known to a caller at the start of the call, it is easy to extend the algorithms in Sections 3 and 4 to route the call using the greedy algorithm. The path simply creates branches where necessary in levels $\lg N$ through $2\lg N$ of the multi-Benes network to reach all the desired output terminals. The bit complexity of the algorithm may increase, however, to reflect the Kolmogorov complexity of the set of outputs that the path must reach.

If parties to a multiparty call are to be added after the call is already underway, then we use a multi-Benes network with wraparound connections, and add parties by routing paths from parties already involved in the call. Each addition takes $O(\log N)$ steps, and by being careful, we can insure that the resulting "depth" of the call is at most $O(\log^2 N)$. This is not quite as elegant as the solutions proposed in [7] for generalized non-blocking networks, but no routing algorithms are known at all for those constructions.

If many parties want to call the same output terminal, then we have two options: merging the callers into a single multiparty call, or giving busy signals to all but one of the callers. Both options can be performed in $O(\log N)$ bit-steps, but both require the use of the AKS sorting circuit for doing some processing. In particular, we use the AKS circuit to sort the calls according to their destination, and then use a prefix operation to combine them or to send busy signals to redundant calls. Calls not receiving busy signals can then proceed as before. Calls that are combined are handled in a manner analogous to the way we handled multiparty calls, only in reverse.

The details of all of these procedures will be included in the final draft of the paper.

6 Acknowledgments

The authors thank Nick Pippenger for suggesting that a nonblocking network might be constructed by treating busy switches as if they were faulty.

References

[1] W. Aiello, T. Leighton, B. Maggs, and M. Newman. Manuscript in preparation.

[2] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(N \log N)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 1–9, April 1983.

[3] L. A. Bassalygo and M. S. Pinsker. Complexity of optimum nonblocking switching network without reconnections. *Problems of Information Transmission*, 9:64–66, 1974.

[4] V. E. Benes. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:1641–1656, July 1964.

[5] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32:406–424, 1953.

[6] D. Dolev, C. Dwork, N. Pippenger, and A. Widgerson. Superconcentrators, generalizers and generalized connectors with limited depth. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 42–51, April 1983.

[7] P. Feldman, J. Friedman, and N. Pippenger. Wide-sense nonblocking networks. *SIAM Journal of Discrete Mathematics*, 1(2):158–173, May 1988.

[8] F. T. Leighton. Parallel computation using meshes of trees. In *1983 Workshop on Graph-Theoretic Concepts in Computer Science*, pages 200–218, Trauner Verlag, Linz, 1984.

[9] T. Leighton and B. Maggs. Expanders might be practical: fast algorithms for routing around faults in multibutterflies. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 384–389, IEEE, October 1989.

[10] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[11] G. A. Margulis. Explicit constructions of concentrators. *Problems of Information Transmission*, 9:325–332, 1973.

[12] N. Pippenger. Personal communication.

[13] N. Pippenger. On rearrangeable and nonblocking switching networks. *Journal of Computer and System Sciences*, 17:145–162, 1978.

[14] N. Pippenger and A. C.-C. Yao. Rearrangeable networks with limited depth. *SIAM Journal of Algebraic and Discrete Methods*, 3:411–417, 1982.

[15] C. E. Shannon. Memory requirements in a telephone exchange. *Bell System Technical Journal*, 29:343–349, 1950.

[16] J. Turner. Personal communication.

[17] E. Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 241–250, May 1989.

Minimization of Functions with Multiple-Valued Outputs: Theory and Applications

Srinivas Devadas

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, Cambridge

Abstract

In this paper, we present a theoretical framework for the minimization of logic functions with symbolic or multiple-valued outputs. Using this framework, we develop efficient, exact algorithms for the problems of output encoding and finite state machine (FSM) state assignment. All previous automatic approaches to these encoding problems have involved the use of heuristic techniques. Other than the straightforward, exhaustive search procedure, no exact solution methods have been proposed.

When targeting two-level logic implementations, the problems of output encoding and state assignment involve a search for appropriate binary codes for the symbolic outputs or states so as to obtain a minimum number of product terms after two-level Boolean minimization. A straightforward, exhaustive search procedure requires $O(N!)$ exact Boolean minimizations, where N is the number of symbolic outputs or states. We define a notion of *generalized prime implicants* for functions with multiple-valued outputs, and propose a novel minimization procedure of prime implicant generation and covering for solving the output encoding problem. An optimum solution to this covering problem is also an optimum solution to the encoding problem. A single logic minimization step thus replaces $O(N!)$ minimizations.

It has been shown previously that the input encoding problem can be exactly solved using Boolean minimization over functions with multiple-valued inputs. An extension of our algorithm that handles functions with multiple-valued inputs and outputs can be used to solve the state assignment problem exactly.

Experimental results are presented on a set of examples.

1 Introduction

Input and output encoding problems in switching theory involve the assignment of binary codes to symbolic inputs and outputs so as to minimize a given cost function, typically, the area of the resulting logic network. In this paper, we are concerned with two-level or Programmable Logic Array (PLA) implementations of logic functions. The cost function targeted in this case is the number of product terms in the eventual two-level implementation. State assignment, one of the oldest problems in automata theory, is also an encoding problem.

The difficulty in optimal encoding stems from the fact that one has to model a complicated logic optimization step that follows. For instance, given the symbolic truth-tables like those in Figure 1, which are to be implemented in PLA form, one wishes to find an encoding of the *inp*1, .., *inpN* (*out*1, .., *outM*) so as to minimize the number of product terms of the resulting PLA *after* two-level Boolean minimization. A simple, exhaustive search technique to find the optimum encoding requires $O(N!)$ ($(O(M!))$ *exact* two-level Boolean minimizations. Two-level Boolean minimization is a mature area – programs like ESPRESSO-EXACT [6] and McBOOLE [2] minimize large functions exactly using reasonable amounts of CPU time. However, the *number* of required minimizations makes an exhaustive search approach to optimum encoding infeasible for anything but the smallest problems.

10	inp1	1010	0001	out1
01	inp1	0110	00-0	out2
10	inp2	1010	0011	out2
-1	inp2	1011	0100	out3
1-	inp3	0110	1000	out3
0-	inp3	1001	1011	out4
--	inp4	0010	1111	out5
--	inp5	1101		

(a) (b)

Figure 1: Symbolic Covers

In [5], it was shown that the input encoding problem, where the objective is to minimize the number of product terms in an eventual two-level implementation, can be solved exactly by means of an exact minimization of a function with multiple-valued inputs. The given symbolic cover is transformed into a function with multiple-valued inputs and an optimum encoding, that results in a minimum binary cover whose cardinality equals the cardinality of the minimized multiple-valued cover is found.

The corresponding problem of optimal output encoding has remained largely unsolved due to the lack of a viable theory for the minimization of logic functions with multiple-valued outputs.

In this paper, we present an *exact algorithm for output encoding, based on the minimization of functions with symbolic/multiple-valued outputs*, rather than binary-valued outputs. The algorithm finds an encoding that minimizes the number of product terms in an optimized two-level implementation. While the algorithm has worst-case exponential complexity, its average-case behavior is significantly better than exhaustive search method, since *a single minimization replaces O(N!) minimizations* (N is the number of symbolic outputs to be encoded). The algorithm consists of the following steps

1. Generation of generalized prime implicants (GPIs) from the original symbolic/multiple-valued output cover.
2. Solution of a constrained covering problem, namely, selection of a minimum number of GPIs that form encodeable cover.
3. Encoding of the symbolic outputs respecting the encoding constraints generated during Step 2.
4. Given the codes of the symbolic outputs and the selected GPIs, a two-level cover with product term cardinality equal to the number of GPIs can be trivially constructed. This two-level cover represents an exact solution to the encoding problem.

Classical prime implicant generation techniques can be modified to generate GPIs in Step 1. The covering problem of Step 2 is more complicated than the classical (unate) covering problem. Hence, classical covering algorithms cannot be directly used. Step 3 involves constrained encoding where the objective is to minimize the number of encoding bits required to satisfy constraints derived from Step 2. This step is also NP-complete. However, our focus here is to exactly minimize product term cardinality and heuristically minimize the area of the PLA that implements the cover.

We have also developed an exact state assignment algorithm that has essentially the same structure as the above procedure. In the state assignment case, the present states are represented as different values of a single multiple-valued variable (as in [5]). The covering problem is more complex than in the output encoding case and so is the constrained encoding problem. Due to space constraints we will concentrate on the output encoding problem in this paper.

In Section 2, basic definitions and notations used are given. The exact output encoding algorithm is described in Section 3. We give theorems that prove the correctness of the procedure. Pruning heuristics that can be used in the exact solution of the covering problem in output encoding are described in Section 4. Techniques for the creation of reduced prime implicant tables are also described. Preliminary experimental results are presented in Section 5.

2 Preliminaries

Let $B = \{0, 1\}$ $Y = \{0, 1, 2\}$. A **logic** (Boolean, switching) **function** ff in n input variables, $x_1, x_2, .. x_n$, and m output variables, $y_1, y_2, .. y_m$, is a function

$$ff : B^n \rightarrow Y^m$$

where $x = [x_1, .. x_n] \in B^n$ is the **input** and $y = [y_1, .. y_m] \in Y^m$ is the **output** of ff. B^n is the Boolean n-space associated with the function ff. Note that in addition to the usual values of 0 and 1, the outputs y_i may also take the **don't care value** 2 (or $-$). Such functions are called **incompletely specified** logic functions. A **completely specified function** f is a logic function taking values in $\{0, 1\}^m$, i.e., all the values of the input map into 0 or 1 for all the components of f. For each component of an incompletely specified logic function ff, ff_i, $i = 1, .. m$, one can define: the **ON-set**, $X_i^{ON} \subset B^n$, the set of input values x such that $ff_i(x) = 1$, the **OFF-set**, X_i^{OFF}, the set of values such that $ff_i(x) = 0$ and the **don't care** set X_i^{DC}, the set of values such that $ff_i(x) = 2$. A logic function with $m = 1$ is called a **single-output** function, while $m > 1$, it is called a **multiple-output** function.

A **cube** in a Boolean n-space associated with a logic function, f, can be specified by its vertices and by an index indicating to which components of f it belongs. An input cube c is specified by a row vector $c = [c_1, .. c_n]$ where each input variable takes on one of three values 0, 1 or 2 (or $-$). A 2 in the cube is a don't care input, which means that the input can take the values of either 0 or 1. For example, the cube 002 is equal to the union of the cubes 001 and 000. A **minterm** is a cube with only 0 and 1 entries. Cubes can also be classified based on the number of 2 entries in the cube. A cube with k entries or bits which take the value 2 is called a **k-cube**. A minterm thus is a **0-cube**.

A cube c_1 is said to **cover** (contain) another cube c_2, if each entry of c_1 is equal to the corresponding entry of c_2 or is equal to 2. A minterm m_1 is said to **dominate** another minterm m_2 (written as $m_1 \supset m_2$) if for each bit position in the second minterm that contains a 1, the corresponding bit position in the first minterm also contains a 1. Minterm m_2 is dominated by m_1 (written as $m_2 \subset m_1$). The **disjunction** of two minterms is the bitwise OR (written as | or $\cup$) of the two minterms. The **conjunction** of two minterms is the bitwise AND (written as $\cap$) of the two minterms.

A logic function may have **multiple-valued** or symbolic input variables and symbolic output variables as in Figure 1. A symbolic input or output variable takes on **symbolic values**.

0001 001	0001 10000
00-0 010	00-0 01000
0011 010	0011 01000
0100 011	0100 00100
1000 011	1000 00100
1011 100	1011 00010
1111 101	1111 00001
(a)	(b)

Figure 2: Possible Encodings of the Symbolic Output

3 Output Encoding

3.1 Introduction

The output encoding problem entails finding binary codes for symbolic outputs in a switching function so as to minimize the area or an estimate of the area of logic function after encoding and logic optimization. Here, we are concerned with two-level or PLA implementations of logic – the optimization step that follows encoding is one of two-level Boolean minimization.

An example encoding of the symbolic output of the function shown in Figure 1(b), is shown in Figure 2(a). The encoded cover is now a multiple-output Boolean function. This function can be minimized using standard two-level Boolean minimization algorithms. These algorithms exploit the sharing of terms between the different outputs so as to produce a minimum or minimal cover. An encoding such as the one in Figure 2(b), where each symbolic value corresponds to a separate output, can have no sharing of terms between the outputs. Two-level Boolean minimization of the cover of Figure 2(b) would produce a cover with a number of product terms equal to the total number of product terms produced by *disjointly* minimizing each of the ON-sets of the symbolic values of Figure 1(b). This cardinality is typically far from the minimum cardinality achievable via an encoding that results in maximal sharing of product terms across outputs.

3.2 Previous Work

Approaches to solving the output encoding problem in the past have involved the use of heuristic techniques (*e.g.* [4, 7])

The program CAPPUCINO [4] attempts to minimize the number of product terms in a two-level implementation and secondarily the number of encoding bits. The algorithm in CAPPUCINO is based on exploiting *dominance* relationships between the binary codes assigned to different values of the symbolic output that is to be encoded. Consider the the example of Figure 1(b). If the symbolic value *out*1 is given a binary code 110 which dominates the binary code 100 assigned to *out*2, then the input cubes corresponding to *out*1 can be used as don't cares for minimizing the input cubes of *out*2. Exploiting these don't cares can reduce the cardinality of the ON-set of the symbolic value *out*2. In CAPPUCINO, an attempt is made to heuristically construct dominance relationships between symbolic values that result in maximal reduction of the ON-sets of the dominated symbolic values. Satisfying a non-conflicting set of dominance relationships results in some re

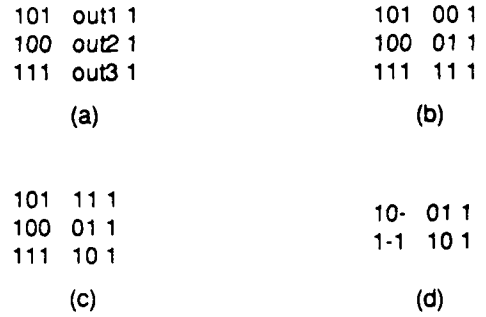

Figure 3: Dominance and Disjunctive Relationships

duction of the overall cover cardinality. There is no guarantee of achieving minimum cardinality because all possible dominance relations are not explored, nor is an optimum set selected. However, a more basic shortcoming is that dominance relations are *not* the only kind of relationships between symbolic values that can be exploited. After a symbolic cover has been encoded, it represents a multiple-output logic function. As we will show, minimizing a multiple-output function entails exploiting other sharing relationships than just dominance.

3.3 Disjunctive Relationships

Consider the symbolic cover of Figure 3(a), that has one symbolic output and one binary-valued output. Using exclusively dominance relationships in an encoding, it is not possible to reduce the size of any of the ON-sets of the symbolic values. One such encoding is shown in Figure 3(b). The binary code 00 has been given to *out*1, 01 given to *out*2 and 11 given to *out*3. However, if we code *out*1 with 11, *out*2 with 01 and *out*3 with 10 as in Figure 3(c), a reduction in cover cardinality after minimization can be obtained (Figure 3(d)). As described earlier, in a dominance relationship, the ON-set of the *dominated* symbolic value is reduced. In Figure 3(c) and 3(d), however, it is in fact the *dominating* symbolic value, *out*1, whose ON-set cardinality has been reduced from 1 to 0. This can be explained by the *disjunctive relationship* between the codes of *out*2, *out*3 and *out*1. We have *out*1 = *out*2 | *out*3 and hence the ON-set of *out*1 can be reduced using the ON-set of *out*2 and *out*3. Simply making *out*1 dominate *out*2 and *out*3 is not enough, the code of *out*1 has to be the disjunction (bitwise OR) of the codes of *out*2 and *out*3. Exploiting these relationships is basic to a multiple-output logic minimizer. Hence, an exact encoding algorithm has to take into account these relationships in order to produce a minimum cardinality cover after optimization. Disjunctive relations may involve an arbitrary number of symbolic values. The code of a symbolic value may be the bitwise OR of two or more symbolic value codes.

Enumerating dominance or disjunctive relationships can be very time-consuming. Furthermore, finding the reduction in cover cardinality that can be accrued via an encoding satisfying each dominance or disjunctive relationship requires an exact logic minimization. Moreover, these relationships interact in complex ways and their effects are not simply cumulative. An efficient, exact solution to the output encoding problem involves the modification of prime implicant generation and covering strategies that are basic to two-level Boolean minimization.

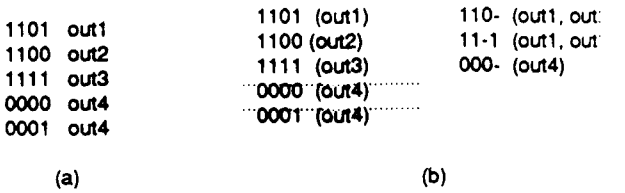

Figure 4: Generation of Generalized Prime Implicants

3.4 An Algorithm for Optimal Output Encoding

In this section, we present an exact algorithm for output encoding that guarantees the minimum number of product terms in an encoded and optimized cover. As described briefly in Section 1, the algorithm consists of four steps. We detail the steps in the remainder of this section.

We are given a symbolic cover S with a single symbolic output (*c.f.* Section 3.6 for generalization to the multiple symbolic output case). The different symbolic values are denoted $v_1, .., v_N$. The ON-sets of the v_i are denoted by C_i. Each C_i is a set of D_i minterms $\{m_{i1}, .. m_{iD_i}\}$. Each minterm m_{ij} has a tag as to what symbolic value's ON-set it belongs to. A minterm can only belong to a single symbolic value's ON-set. Minterms are also referred to as 0-cubes.

3.4.1 Generating Generalized Prime Implicants

The generation of generalized prime implicants (GPIs) proceeds in a manner similar to the well-known Quine-McCluskey (Q-M) procedure [3], with some differences.

1-cubes are composed by merging all pairs of mergeable cubes. If two 0-cubes with the same tag, namely, (v_i), are merged then the 1-cube has the same tag (v_i). If a 0-cube of tag (v_i) is merged with a 0-cube with tag (v_j), the resultant cube has a tag (v_i, v_j). The rule for canceling 0-cubes cover by 1-cubes is different from the Q-M method. A 0-cube can be canceled by a 1-cube if and only if the 1-cube covers the 0-cube *and* their tags are identical. For example, a 1-cube 1- with tag (v_1, v_2) cannot cancel a 0-cube 110 with tag (v_1).

The above can be generalized to k-cubes.

1. When two k-cubes merge to form a $k+1$-cube, the tag the $k+1$-cube is the *union* of the two k-cube tags.
2. A $k+1$-cube cancels a k-cube only if the $k+1$-cube covers the k-cube and they have identical tags.

A cube with a tag that contains all the symbolic values $(v_1, .., v_N)$ can be discarded and is not a GPI. These cubes are not required in a minimum solution (*c.f.* Theorem 3.3). The generation of generalized prime implicants for the symbolic cover of Figure 3(a) is illustrated in Figure 4. We have 6 GPIs with associated tags.

3.4.2 Selecting a Minimum Encodeable Cover

Given all the GPIs, we have to select a minimum subset such that they cover all the minterms and form an *encodeable* cover. The difference between the output encoding problem and two-level Boolean minimization is this additional restriction of encodeability for a selected subset of GPIs. The selection is performed by solving a covering problem (*c.f.* Section 4). In the sequel, we describe the meaning of an encodeable cover.

Consider a minterm, m, in the original symbolic cover S, belonging to the ON-set of v_m. Obviously, in any encoded cover the minterm m has to assert the code given to v_m, namely

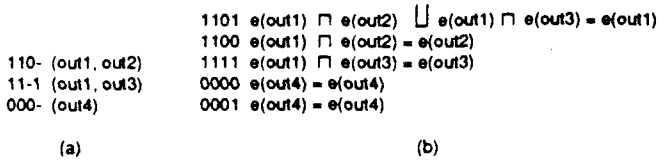

Figure 5: Encodeability of Selected GPIs

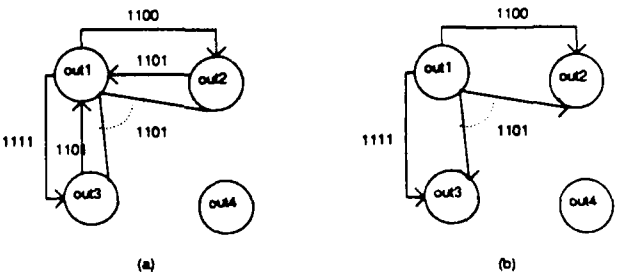

Figure 6: Encodeability Graphs

$e(\ v_m\ )$. Let the selected subset of GPIs be p_1, .., p_G and let the GPIs that cover m in this selected subset be $p_{m,1}$, .., $p_{m,M}$. For functionality to be maintained, for all minterms $m \in S$, the following relation has to be satisfied:

$$\bigcup_{i=1}^{M} \bigcap_{j} e(\ v_{p_{m,i},j}\ ) \quad = \quad e(\ v_m\ ) \quad \forall\, m \qquad (1)$$

where the $v_{p_{m,i},j}$ represent the symbolic values that are contained in the tag of the GPI $p_{m,i}$. In Figure 5, we have a selection of GPIs for the symbolic cover of Figure 4(a) (all of whose GPIs are shown in Figure 4(b)). We have selected three GPIs $110-$, $11-1$ and $000-$ from Figure 4(b) in Figure 5(a). The constraints corresponding to Eqn. 1 for each minterm are shown in Figure 5(b). The minterm 1101 is covered by two selected GPIs, one with a tag ($out1$, $out2$) and the other with a tag ($out1$, $out3$). Therefore, Eqn. 1 specifies:

$$e(out1) \cap e(out2) \bigcup e(out1) \cap e(out3) \quad = \quad e(out1)$$

for the minterm 1101. Other constraints are specified for the remaining minterms. If a minterm is covered by a single GPI with the same tag as the minterm, then the constraint specified by the minterm via Eqn. 1 is an identity.

Eqn. 1 specifies a set of constraints on the codes of the symbolic values, given a selection of GPIs forming a cover. If an encoding can be found that satisfies all these constraints, then the cover is encodeable. However, a cover may have an associated set of constraints that are mutually conflicting.

3.4.3 Dominance and Disjunctive Relationships to Satisfy Constraints

The constraints specified by Eqn. 1 can be satisfied by means of disjunctive and dominance relations between symbolic values. Continuing with our example, to satisfy:

$$e(out1) \cap e(out2) \bigcup e(out1) \cap e(out3) \quad = \quad e(out1)$$

one has three alternatives:

1. $e(out2) \supset e(out1)$
2. $e(out3) \supset e(out1)$
3. $e(out1) \subseteq e(out2) \mid e(out3)$

Given an arbitrary constraint, a set of dominance and disjunctive relations can be derived such that satisfying any single relation satisfies the constraint. It is possible that dominance and disjunctive relationships conflict across a set of constraints. For example, one cannot satisfy both $e(out1) \supset e(out2)$ and $e(out2) \supset e(out1)$. This can be represented as a cycle in the set of dominance relations. Also, if one picks the equality in choice (3) above, then that implies $e(out1) \supset e(out2)$ and $e(out1) \supset e(out3)$. In that case, one cannot satisfy both (1) and (3) simultaneously.

Given a selection of GPIs, we derive a set of constraints via Eqn. 1 and construct a graph where each node corresponds to a symbolic value. Directed edges in the graph represent dominance relations and undirected edges enclosed by (dotted) arcs represent disjunctive relations. Each directed edge and arc has a label, corresponding to the minterm producing the constraint represented by the edge or arc. The graph corresponding to the selected set of GPIs of Figure 5(a) is shown in Figure 6(a). A directed edge from $out1$ to $out2$ implies that the code of $out1$ should dominate the code of $out2$. The dotted arc around the two undirected edges emanating from $out1$ implies that the code of $out1$ should be equal to, or be dominated by, the disjunction (bitwise OR) of the codes of its fanout symbolic values, *i.e.* $out2$ and $out3$. We have $e(out1) \subseteq e(out2) \mid e(out3)$. $out1$ is called the **parent** in the disjunctive arc and $out2$ and $out3$ are the **siblings** in the disjunctive arc. The disjunctive arc specifies equality or dominance, however, due to other relationships equality may be specifically required. In the case of disjunctive dominance the edges are undirected, in the case of disjunctive equality the edges are directed towards the siblings to indicate that the parent dominates the siblings.

The graph corresponding to a selection of GPIs is encodeable and logic functionality is maintained, if four conditions are met. For each label, one selects either an edge or an arc with that label. In the case of selecting an arc, all dominance edges covered by the arc (implied by the disjunctive relationship) are also selected. For some selection,

1. There should be no directed cycles in the graph.
2. The siblings in any disjunctive arc should not have directed paths between each other.
3. No two disjunctive equality arcs can have exactly the same siblings and different parents.
4. The parent of a disjunctive dominance (equality) arc should not dominate (any symbolic value/node that dominates) all the siblings in the arc.

The graph of Figure 6(b), derived from the graph of Figure 6(a), satisfies these properties and hence the selection of GPIs is valid. This implies that we can find an encoding such that the optimized cover has 3 product terms.

Given a constraint specified by Eqn. 1 of the form

$$a \cap b \cap c \bigcup a \cap d \cap e \bigcup a \cap f \cap g \quad = \quad a \qquad (2)$$

we have more complex choices than in the equation of Figure 5(b). To satisfy Eqn. 2, we can satisfy $a \cap b \cap c \quad = \quad a$. In

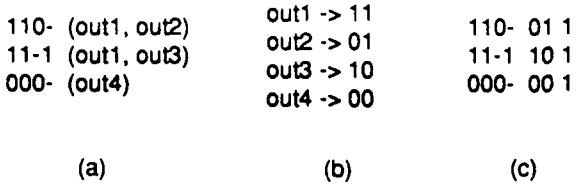

Figure 7: Constructing the Optimized Cover

this case, we need to satisfy both $b \supset a$ and $c \supset a$. This corresponds to a pair of directed edges that have to be selected simultaneously. One can also satisfy $a \cap b \cap c \bigcup a \cap d \cap f = a$ by satisfying $b \cap c \bigcup d \cap f \supseteq a$. This corresponds to a disjunctive relationship with nested conjunctive terms. The siblings $b \cap c$ and $d \cap f$ are called **conjunctive nodes**. These conjunctive nodes are dominated by b, c and d, f respectively. Conditions 2-4 are required to be satisfied for arcs whose siblings are conjunctive nodes as well. The symbolic values whose conjunction forms the conjunctive node are called the **ancestors** of the node. As mentioned earlier, the ancestors dominate the conjunctive node. If all the ancestors dominate a particular symbolic value, then the conjunctive node also dominates that value, which can have some interesting effects. For example, if we have all the ancestors of a conjunctive node dominating the parent of a disjunctive arc that the node is a sibling of, then we have a cycle in the graph rendering it unencodeable.

3.4.4 Constructing the Optimized Cover

If a selection of GPIs has been made that covers all minterms and is encodeable, then an encoding can be found that satisfies the constraints (see Theorem 3.4). We can now construct an encoded and optimized cover directly. The cover will contain exactly the selected GPIs. For each GPI, the output combination in the cover is found by inspecting the tag corresponding to the GPI. The codes corresponding to all the symbolic values contained in the tag of the GPI are intersected (bitwise ANDed) to produce the output part. Continuing with our example, the GPIs selected with associated tags for the GPIs are shown in Figure 7(a). These GPIs have an associated graph that is encodeable (Figure 6(b)) and an encoding satisfying the constraints is given in Figure 7(b). The encoding satisfies disjunctive equivalence $e(out1) = e(out2) \mid e(out3)$, rather than disjunctive dominance $e(out1) \subset e(out2) \mid e(out3)$. This is because the dominance relationships $e(out1) \supset e(out2)$ and $e(out1) \supset e(out3)$ have to be satisfied. In Figure 7(c), we have constructed the optimized cover with the GPIs by intersecting the codes of symbolic values in the tags of each GPI to obtain the output part.

3.5 Correctness of Procedure

Proposition 3.1 *The selection of a minimum cardinality encodeable cover from the GPIs represents an optimum solution to the output encoding problem.*

In the sequel, we will justify Proposition 3.1. First, we show that logic functionality is retained via the procedure.

Lemma 3.2 *Satisfying Eqn. 1 and constructing the output part as in Section 3.4.4 retains logic functionality.*

Proof: We construct the output part of a GPI by intersecting all the codes of the symbolic values in the GPI's tag. That corresponds precisely to the intersection ($\cap$) term in Eqn. 1. The output of a minterm in a two-level cover is the disjunction of all the outputs asserted by the cubes that cover the minterm. This corresponds to the union ($\cup$) in Eqn. 1. Thus, satisfying Eqn. 1 guarantees that each minterm asserts the same output combination as it would have in the original encoded, but unoptimized, cover. ∎

We now show that the canceled k-cubes during GPI generation are not necessary in a minimum solution.

Theorem 3.3 *A minimum cardinality, encodeable cover can be made up entirely of GPIs.*

Proof: Assume that we have a minimum cardinality encodeable cover with a cube c_1 that is not a GPI. Let the tag of c_1 be T. We know a GPI p_1 exists such that $p_1 \supset c_1$ and that the tag of p_1 is T. We replacing c_1 with p_1 in the cover. This obviously does not change the cardinality of the cover. The minterms contained in $p_1 - c_1$ will be covered by an extra GPI p_1 and therefore Eqn. 1 for these minterms will be different. However, the extra term in the equation (added to the union) merely represents an extra option in the graph corresponding to the encodeability. Since the original graph was encodeable, adding edges with the same label as the labels of edges originally in the graph does not change the encodeability.

During GPI generation, we discard cubes with tags that contain all the symbolic values. Say that such a cube exists in a minimum encoded cover. Then, it asserts the output combination given by the intersection of the codes of *all* the symbol values. If this intersection is null (all 0s), then the cube can be discarded and a smaller cover with the same functionality can be obtained. If the intersection is not null and the cube asserts some outputs, then it means that in these bits corresponding to these outputs, *all* the codes of the symbolic values have a We can reduce the codes of all the values, while keeping them distinct, by discarding these outputs. The cube then asserts a null output combination and can be discarded. Thus, the cube is not required in a minimum cover.

Hence, we have a minimum cardinality encodeable selection that is made up entirely of GPIs. ∎

Thus, if one selects a minimum set of GPIs that cover all minterms and have an associated set of constraints by Eqn. 1 that is encodeable, a minimum solution to the encoding problem is guaranteed. It remains to be proven that the conditions to the satisfied by the graph for encodeability are necessary and sufficient. The proof of the following theorem is given in the Appendix.

Theorem 3.4 *Conditions 1-4 stated in Section 3.4.3 are necessary and sufficient for the graph to be encodeable.*

3.5.1 The Issue of the All Zeros Code

If a code of all zeros is given to a symbolic value, then it is possible that one or more GPIs can be dropped in a two-level implementation, from an otherwise minimum cover. This is because in a two-level implementation, we are only concerned with the ON-sets. The procedure presented thus far has not taken this fact into account.

A solution is to perform $N + 1$ minimizations, N being the number of symbolic values. The first minimization is as before. In the other N minimizations, we drop all the minterms in the ON-set of each of the N symbolic values, one value's ON-set at a time. We select the best solution resulting from the $N + 1$ minimizations. The reason the first minimization has to performed without dropping any of the minterms is that the all zeros code cannot appear in disjunctive relations, since it is dominated by all other codes. Hence, constraining ones not to use a code of all zeros may result in a sub-optimal solution.

We can prove the following theorem which gives a condition when multiple minimizations are not required.

Theorem 3.5 *Given a cover with one or more symbolic outputs and binary-valued outputs if all minterms in the cover belong to the ON-set of at least one binary-valued output, then there can be no advantage to using an all zeros code.*

Proof: When using an all zeros code, the only advantage accrued is that minterms may be dropped by putting them into OFF-sets. Any set of dominance and/or disjunctive relationships can be satisfied via codes other than the all zeros code. In the case of a cover with the property described in the theorem, we cannot drop any of the minterms. Hence, for such a cover, we can obtain a minimum cardinality solution via a single minimization and without using the all zeros code. ■

3.6 Multiple Symbolic Outputs

The procedure outlined can be generalized to the case where we have multiple symbolic outputs, all of which have to be encoded. Each minterm initially has a number of tags equal to the number of symbolic outputs. For each symbolic output, the tag corresponds to the symbolic value whose ON-set the minterm belongs to. Minterm pairs are merged and the operations on the tags are performed exactly as before. A $k+1$-cube cancels a k-cube if and only if all of its tags are identical to the corresponding tags of the k-cube. Cubes with tags such that all the corresponding symbolic values, for all symbolic outputs, are contained in each tag can be discarded. Thus, the GPIs can be generated. We have separate graphs representing the encoding constraints for each symbolic output. Given a selection of GPIs, these graphs are constructed and checked for encodeability exactly as before. For a selection of GPIs to be valid, all the graphs have to be encodeable.

The procedure can be easily generalized to functions with both symbolic and binary-valued outputs.

4 Solving the Covering Problem

4.1 Introduction

The classical covering problem of two-level Boolean minimization involves the selection of a minimum set of prime implicants (PIs) that form a cover for a logic function. In output encoding, we have an additional restriction on the selected generalized prime implicants (GPIs) in that they have to form an *encodeable* cover.

In Section 3.4.1, we first describe how techniques that generate all the the prime implicants of binary-valued output functions can be used to generate all the GPIs for functions with multiple-valued outputs. In Section 4.3, we review strategies for solving the covering problem in two-level Boolean minimization. In Section 4.4, we describe our approach to solving the covering problem with associated encodeability constraints.

4.2 Reduced Prime Implicant Table Generation

Many techniques for determining all the prime implicants (PIs) of single and multiple-output logic functions have been developed in the past (*e.g.* [3] [8]). An algorithm based on the recursive decomposition of a function followed by a pairwise consensus operation has been reported [1]. This algorithm was improved upon in the program McBOOLE [2]. Other techniques have been reported as well [6]. These techniques not only efficiently generate PIs without duplication of effort but also create what is called a *reduced prime implicant table*. The prime implicant table of the Q-M algorithm is such that each column in the table corresponds to a minterm of the function and each row to a PI. In a reduced prime implicant table, however, each column corresponds to a collection of minterms (*i.e.* a larger subspace), all of which are covered by the same set of PIs. Thus, using the algorithms of [6] for example, rather than the Q-M method can lead to a much more efficient creation of the prime implicant table.

We cannot directly use techniques such as those of [6] on functions with multiple-valued outputs to generate all GPIs.

0001 out1	0001 11110
00-0 out2	00-0 11101
0011 out2	0011 11101
0100 out3	0100 11011
1000 out3	1000 11011
1011 out4	1011 10111
1111 out5	1111 01111
(a)	(b)

Figure 8: Transformation for Output Encoding

The canceling rule for GPIs (*c.f.* Section 3.4.1) is not the same as the canceling rule for PIs. However, as we will show, we can transform a function with a multiple-valued output into a function with multiple binary-valued outputs such that the PIs for this new multiple-output function have a one-to-one correspondence with the GPIs of the original function. In Figure 8, the function with a symbolic/multiple-valued output of Figure 1(b) has been duplicated in Figure 8(a). Each symbolic value has been replaced by an output combination to produce the binary-valued multiple-output function of Figure 8(b). The number of outputs is equal to the number of symbolic values. Each symbolic value has an output combination of all 1s except for one 0 in a unique identifying position. These outputs perform the function of the output tags in GPI generation.

Lemma 4.1 *The PIs of the function obtained via the transformation of Figure 8 are the GPIs of the original function with the symbolic output.*

Proof: The set of outputs asserted by any cube in the new binary-valued multiple-output function is the set of symbolic values *not* in the tag of the corresponding cube in the original function. During PI generation for the binary-valued function, a cube, c_1, cancels another cube, c_2, only if c_1 covers c_2 and the outputs asserted by c_1 are the same as the outputs asserted by c_2. This implies that the set of symbolic values in the tag of the two corresponding cubes in the original function are identical and c_1 would have canceled c_2 during GPI generation. Finally, cubes in the binary-valued function with a null output combination are discarded. This corresponds to discarding cubes with tags that contain all the symbolic values. ■

Thus, via this transformation we can make use of efficient techniques for prime implicant generation to generate GPIs. Once the GPIs have been generated, the added outputs are discarded, since we have to solve a different covering problem from the classical covering problem. The output tags for each GPI are constructed by finding all the symbolic values whose ON-sets intersect the GPI.

4.3 The Classical Covering Problem

A branch-and-bound solution [6] to the minimum cover problem involves the following steps (columns correspond to collections of minterms and rows correspond to PIs):

1. Remove columns that contain other columns and remove rows which are contained by other rows. Detect essential rows – a column with a single 1 identifies an essential row – and add these to the selected set. Repeat until no new essential elements are detected.

2. If the cardinality of the selected set exceeds the best solution thus far, return from this level of recursion. If there

are no elements left to be covered, the selected set is the best solution recorded thus far.

3. Heuristically select a branching PI, *i.e.* row.

4. Add this row to the selected set and recur for the subtable that results from deleting the row and all columns that are covered by this row. Then, recur for the subtable that results from deleting this row without adding it to the selected set.

In [6], a lower bounding strategy based on a maximal independent set heuristic was proposed. In Step 2, a maximal set of columns, all of which are pairwise disjoint is found using a straightforward, greedy algorithm [1]. Since each column must be covered and all the columns in the maximal independent set do not share any row(s), the cardinality of the maximal independent set is a lower bound on the number of rows required to complete the cover. At Step 2, the recursion is bounded if the cardinality of the selected set at Step 2 *plus* the cardinality of the maximal independent set equals or exceeds the current best solution.

4.4 Covering with Encodeability Constraints

The covering algorithm for output encoding is a modification of the algorithm described in the previous section. The required modifications are described in the sequel.

In Step 1, a row (GPI) is deemed to contain another row (GPI) only if (1) the tags of the two GPIs are identical or (2) the tag of the first GPI is a subset of the tag of the second. Condition 2 cannot occur in the initial table, but may occur lower in the recursion after some columns have been deleted. The lower bounding criterion at Step 3 uses the cardinality of the maximal independent set of columns. This bound is looser than in classical covering because even if a cover can be constructed with a number of elements equal to the lower bound, it may not be encodeable.

After obtaining a selected set that covers all elements, we perform an encodeability check. If the cover is encodeable, the solution is declared as the best recorded until then. If not, another branch-and-bound step is performed to find the *minimum* number of GPIs (rows) which when added to the selected set renders it encodeable. We are only concerned with making the cover encodeable in this step, since all the minterms have already been covered. GPIs during this branch-and-bound step are selected from the current sub-table in the recursion. We now describe the second branch-and-bound step.

1. If the selected set is encodeable, then the selected set is declared as the best encodeable solution thus far. If not, the cardinality of the selected set *plus* a lower bound on the required number of rows to produce an encodeable set is checked to see if it equals or exceeds the best encodeable solution obtained thus far. If so, return from this level of recursion.

2. Heuristically select a branching GPI *i.e.* row.

3. Add this row to the selected set, and recur for the subtable that results from deleting this row. Then, recur after deleting the row without adding it to the current set.

A lower bound on the number of GPIs required to render the graph encodeable is computed by finding the number of *disjoint* violations of the encodeability conditions of Theorem 3.4.

Consider the case of there being two cycles in the graph such that the edges in cycle A have different labels from all the edges in cycle B. Further, no unselected GPI should exist that contains both minterms corresponding to the labels of any pair of edges in cycle A and B. In this case, two GPIs are required to break both cycles – these two cycles are deemed disjoint. Similarly, assume we have two separate instances of directed paths between the siblings of a disjunctive arc. Assume that the two sets of edges in the two paths have disjoint sets of labels, and no unselected GPI exists that covers the pair of minterms corresponding to any pair of edges in the two paths. Then, two GPIs are required to remove the two violations. We can also have disjoint violations of Conditions 3 and 4 of Theorem 3.4.

The heuristic selection of a GPI to add to the selected set in Step 2 is performed by selecting a GPI that covers a maximum number of minterms corresponding to the labels of edges that are involved in violations of the encodeability conditions.

[1] Finding a *maximum* independent set of columns is itself NP-complete.

EX	inp	min	val	out	gpi	prod	enc	CPU time
ex1	2	4	4	1	6	3	2	0.1m
ex2	4	15	6	1	23	6	3	0.9m
ex3	6	44	16	2	194	14	6	10.4m
ex4	8	113	20	0	950	50	9	53.6m
ex5	10	213	20	1	8807	-	-	> 1h
ex6	12	410	32	0	$> 10^4$	-	-	-

Table 1: Results Using Output Encoding Algorithm

5 Experimental Results

We present preliminary experimental results obtained on set of examples. In our current implementation, GPIs are generated via the procedures of Section 3.4.1.

The results obtained using the output encoding algorith are summarized in Table 1. The number of inputs to t function (inp), the number of minterms in the original function (min), the number of symbolic values (val), the number of binary-valued outputs (out), the number of GPIs generate (gpi), the number of product terms in the minimized resu (prod), the number of encoding bits (enc), and the CPU time in minutes required for GPI generation, covering and encoding on a microvax-III (CPU time) are given for each example. T covering problem could not be solved in less than a CPU-ho for example **ex5**. All the GPIs could not be generated due to memory limitations for example **ex6**. However, examples **ex3** and **ex4** which have upto 20 symbolic values have be optimally encoded. An exhaustive search method is not viat for these examples.

Using the transformations of Section 4.2 prior to prime implicant generation will increase the size of the examples th can be handled, since a memory-efficient reduced prime imp cant table can be directly constructed.

6 Conclusions

In this paper, we presented an exact algorithm for the proble of output encoding.

The procedure described is, on the average, much more efficient than a straightforward, exhaustive search procedure to solve this problem. A novel minimization procedure of prir implicant generation and covering that operates on multip valued outputs, rather than binary-valued outputs, was used to solve the encoding problem.

Preliminary experimental results indicate that this a proach is viable for larger circuits than an exhaustive sear method. Computationally efficient heuristic approaches based on the exact algorithms are a subject of current research.

7 Acknowledgements

The interesting discussions with Pranav Ashar, Robert Brayton, Kurt Keutzer, Richard Newton, Tony Ma, Alberto Sangiovanni-Vincentelli and Wayne Wolf on encoding problems are acknowledged. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

APPENDIX

A Proof of Theorem 3.4

Proof: *Necessity*: If the graph is cyclic, then we have two dominance relations $v_1 \supset v_2$ and $v_2 \supset v_1$ that cannot be simultaneously satisfied. If two siblings v_1 and v_2 of a disjunctive arc have a directed path between each other, then either $v_1 \supset v_2$ or $v_2 \supset v_1$. We require $v_1 \mid v_2 = v_3$. If $v_1 \supset v_2$ ($v_2 \supset v_1$) then $v_1 = v_3$ ($v_2 = v_3$), which violates the condition that all codes have to be distinct. If two disjunctive equality arcs with exactly the same siblings exist, the two parents are required to have the same code to satisfy both equalities, which is again a violation. In a graph that violates Condition 4, the conjunction of the siblings in the arc is dominated by some symbolic value that is dominated by the parent. This means that the parent has to both dominate and equal the conjunction of the siblings in a disjunctive equality arc. In a disjunctive dominance arc, the conjunction of the siblings cannot dominate the parent.

Sufficiency: Given a graph that satisfies conditions 1-4, the procedure below constructs a correct encoding. The procedure assumes all disjunctive arcs are binary, but can be easily generalized to multiple siblings in arcs.

We set the codes of all values to null, initially. For each disjunctive arc, the siblings are coded with 01 and 10 and the parent with 11, except for the cases 7 and 8, where more bits may be used. For all other values (or nodes):

1. If the node dominates a single sibling, the sibling's code is appended to the node.
2. If the node dominates both siblings, the parent's code is appended to the node.
3. If the node dominates the parent, the parent's code is appended to the node.
4. If the parent dominates the node, the parent's code is appended to the node.
5. If a single sibling dominates the node, the code of the sibling is appended to the node.
6. If both siblings dominate the node, the code of all zeros is appended to the node.
7. If a node dominates a single sibling and is dominated by the parent, then the following steps are performed. The node and the sibling it doesn't dominate are appended with 101, the dominated sibling with 100 and the parent with 111. In the case of K nodes forming a chain from parent to a sibling, $K+2$ bits will be used, the parent will be coded with $K+2$ 1s, topmost node (level 1) and non-dominated sibling with $K+1$ 1s, a node at level i with $K+2-i$ 1s and the dominated sibling with one 1. By Condition 4, a node cannot dominate both siblings and be dominated by the parent.
8. In the case of a sibling being a conjunctive node, we have to code the ancestors correctly. If one of the ancestors is dominated by the parent, we have Case 7. If both ancestors are dominated by the parent, then the parent is coded with 1111, one ancestor with 1010, the other with 0110 and the sibling gets the intersection 0010. The other sibling's ancestors are coded such that the siblin gets the code 1101. If neither ancestor is dominated by the parent, then one of them is coded with 101, the other with 100, the sibling gets the intersection 100 and the ancestors of the other sibling are given codes such that the other sibling gets the code 011.

It was assumed in the procedure that all disjunctive arcs were disjunctive equality arcs. In the case of an arc being required to be disjunctive dominance, a 1 is appended to a subset of the siblings and a 0 to the parent (without violating other dominance relations). Next, for each disjunctive arc, a composite node is formed by merging the siblings, parent, all nodes dominated by the parent and dominating a sibling and all ancestors of conjunctive node siblings. The graph is then levelized. Let L be the number of levels in the graph. The topmost set of nodes is appended a code of length L with L 1s, the next level codes with $L-1$ 1s and so on. The implication in appending a code to a composite node is that all the nodes contained in the composite node are appended with the same code. ■

References

[1] R. K. Brayton, G. D. Hachtel, Curt McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[2] M. Dagenais, V. K. Agarwal, and N. Rumin. McBOOLE: A Procedure for Exact Boolean Minimization. In *IEEE Transactions on Computer-Aided Design*, pages 229–237, January 1986.

[3] E. J. McCluskey. Minimization of boolean functions. In *Bell Lab. Technical Journal*, pages 1417–1444, Bell Lab., November 1956.

[4] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level macros. In *IEEE Transactions on Computer-Aided Design*, pages 597–616, September 1986.

[5] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State assignment of Finite State Machines. In *IEEE Transactions on Computer-Aided Design* pages 269–285, July 1985.

[6] R. Rudell and A. Sangiovanni-Vincentelli. Exact Minimization of Mutiple-Valued Functions for PLA Optimization. In *Proc. IEEE Int. Conf. on Computer-Aided Design* pages 352–355, 1986.

[7] A. Saldanha and R. H. Katz. Pla optimization using output encoding. In *Proc. of Int'l Conference on Computer Aided Design*, pages 478–481, November 1988.

[8] P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. In *IEEE Transactions on Computers*, pages 446–450, August 1967.

A Fast Multipole Algorithm for Capacitance Extraction of Complex 3-D Geometries

K. Nabors, J. White
Research Laboratory of Electronics
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

December 21, 1989

Abstract

In this paper a fast algorithm for computing the capacitance of a complicated 3-D geometry of ideal conductors in a uniform dielectric is described. The method is an acceleration of the standard integral equation approach for multiconductor capacitance extraction. These integral equation methods are slow because they lead to dense matrix problems which are typically solved with some form of Gaussian elimination. This implies the computation grows like n^3, where n is the number of tiles needed to accurately discretize the conductor surface charges. In this paper we present a preconditioned conjugate-gradient iterative algorithm with a multipole approximation to compute the iterates. This reduces the complexity so that accurate multiconductor capacitance calculations grow as nm where m is the number of conductors.

1 Introduction

In the design of high performance integrated circuits, there are many cases where accurate estimates of the capacitances of complicated three dimensional structures are important for determining final circuit speeds or functionality. Two examples are complicated three-dimensional dynamic memory cells and the three-dimensional chip carriers commonly used in mainframe computers. In these problems, capacitance extraction is made tractable

by assuming the conductors are ideal, and are embedded in a piecewise-constant dielectric medium. Then to compute the capacitances, Laplace's equation is solved numerically over the charge free region with the conductors providing boundary conditions.

Although there are a variety of numerical methods that can be used to solve Laplace's equation, the technique that is typically used in three dimensions is the integral equation approach[ruehli73, rao84, ning88]. In this approach, the surfaces or edges of all the conductors are broken into small tiles. It is assumed that on each tile i, a charge, q_i, is uniformly or linearly distributed. The potential on each tile is then computed by summing the contributions to the potential from all the tiles using Laplace's equation Green's functions. In this way a matrix of potential coefficients, P, relating the set of n tile potentials and the set of n tile charges is constructed, and must be solved to compute capacitances. Typically, Guassian elimination or Cholesky factorization is used to solve the equation, in which case the number of operations is order n^3. Clearly, this approach becomes computationally intractable if the number of unknowns exceeds several hundred, and this limits the size of the problem that can be analyzed to one with a few conductors.

In this paper we present an algorithm for computing capacitance whose complexity grows as mn, where m is the number of conductors. Our algorithm, which is really the pasting together of three well-known algorithms [rohklin86], is presented in three sections. To begin, in the next section one of the standard integral equation approaches is briefly described, and it is shown that the algorithm requires the solution of an $n \times n$ dense symmetric matrix. Then, in Section 3, a preconditioned conjugate-gradient algorithm is described, and it is shown to reduce the complexity of the calculation to order mn^2. In Section 4, it is shown that the conjugate-gradient algorithm only requires the evaluation of a potential field from a charge distribution, and this can be computed in order n time using a multipole algorithm. In Section 5, some preliminary experimental results are given, and we present our conclusions and acknowledgments.

2 The Integral Equation Approach

Consider a system of m ideal conductors embedded in a uniform lossless dielectric medium. For such a system, the relation between the m conductor potentials, denoted by $\hat{p} \in \Re^m$, and the m total charges on each conductor,

denoted by $\hat{q} \in \Re^m$, is given by $\hat{q} = C\hat{p}$, where $C \in \Re^{m \times m}$ is referred to as the capacitance matrix. The i^{th} column of C can be calculated by solving for the total charges on each of the conductors when the i^{th} conductor is at unit potential, and all the other conductors are at zero potential. Then the charge on conductor j, $\hat{q}_j$, is equal to C_{ij}.

There are a variety of approaches for numerically computing the conductor charges given a set of conductor potentials, and we will focus on integral equation methods[ruehli73, rao84, ning88], as they are efficient when applied to problems with ideal conductors in a uniform dielectric medium. The method exploits the fact that the charge is restricted to the surface of the conductors, and rather than discretizing all of free space, just the surface charge on the conductors is discretized. The potential is related to the discretized surface charge through integrals of a Green's functions.

Let the surfaces of a collection of m conductors in free space be discretized into a total of n tiles. The potential at the center of the i^{th} tile would be the sum of the contributions to the potential from the charge distribution on every tile. That is,

$$p_i = \sum_{j=1}^{n} \int_{tile_j} \frac{q_j(r)}{|r - \tilde{r}_i|} da \tag{1}$$

where $\tilde{r}_i$ is the position of the center of tile i, r is the position on the surface of tile j, p_i is the potential at $\tilde{r}_i$, $q_j(r)$ is the position dependent charge density on the surface of the j^{th} tile, and $|r|$ denotes the Euclidian length of r. Note that the integral in (1) is the free space Green's function multiplied by the charge density, integrated over the surface of the j^{th} tile, and that as the distance between tile i and tile j becomes large compared to the surface area of tile j, the integral reduces to $\frac{q_j}{|\tilde{r}_j - \tilde{r}_i|}$ where q_j is the total charge on tile j.

There are several approaches to simplifying (1), the simplest is the "point-matching" approximation in which it is assumed that the charge is distributed uniformly on the tile surface[rao84]. In that case (1) can be simplified to

$$p_i = \sum_{j=1}^{n} \frac{q_j}{a_j} \int_{tile_j} \frac{1}{|r - \tilde{r}_i|} da \tag{2}$$

where q_j is the total charge on tile j, and a_j is the surface area of tile j. When applied to the collection of n tiles, a dense linear system results,

$$Pq = p \tag{3}$$

where $P \in \Re^{n \times n}$; $q, p \in \Re^n$ and

$$P_{ij} = P_{ji} = \frac{1}{2}\left[\frac{1}{a_j}\int_{tile_j} \frac{1}{|r - \bar{r}_i|} da + \frac{1}{a_i}\int_{tile_i} \frac{1}{|r - \bar{r}_j|} da\right]. \quad (4)$$

Note that q and p are the vectors of *tile* charges and potentials rather than the *conductor* charge and potential vectors, $\hat{q}$ and $\hat{p}$ mentioned above. In (4), the potential coefficients, P_{ij}, have been "symmetrized" by averaging for several reasons: the physical system is symmetric, the symmetrized equations have been shown to produce more accurate results for a given discretization, and a symmetric matrix problem is more easily solved. The dense linear system of (3) can be solved, typically by some form of symmetric Gaussian elimination, to compute tile charges from a given set of tile potentials. To compute the j^{th} column of the capacitance matrix, (3) must be solved for q, given a p vector whose entries p_i are set equal to one if tile i is on the j^{th} conductor, and zero otherwise. Then the ij^{th} term of the capacitance matrix is computed by summing all the charges on the j^{th} conductor, i.e. $C_{ij} = \sum_{k \in Conductor_j} q_k$.

3 Using Preconditioned Conjugate-Gradient

In order to solve for a complete $m \times m$ capacitance matrix, the $n \times n$ symmetric matrix of potential coefficients, P, must be factored once, usually into $P = LL^T$, and this requires order n^3 operations. Then, as there are m conductors, the factored system must solved m times with m different right-hand sides, and this requires order mn^2 operations. Since n is the total number of tiles into which the conductor surfaces are cut, m is necessarily much less than n. Therefore, the n^3 factorization dominates for large problems.

This suggests that iterative methods might be more efficient than direct factorization for solving the m charge distribution problems. In particular, as the matrix is symmetric and positive definite, the conjugate-gradient (CG) algorithm is a natural choice[golub83]. Unfortunately, the CG algorithm can converge slowly when applied to the matrix of potential coefficients, particularly when the problem contains widely separated pairs of very closely spaced tiles. To accelerate the convergence of CG, an attempt is made to factor most of the part of the problem associated with the closely spaced tiles directly. To accomplish this, the smallest cube containing the

entire problem is uniformly divided into a large number of cubes, typically into as close to $\frac{n}{10}$ cubes as possible. The piece of the potential coefficient matrix associated with the tile interactions inside a cube is then factored directly and used as a preconditioner to accelerate the CG algorithm. If the p and q vector in (3) are reordered so that tiles contained in a given cube are ordered contiguously, the potential coefficients representing the interaction between tiles in a given cube will be blocks on the diagonal of P. That is, $P = P_{intracube} + P_{intercube}$ where $P_{intracube}$ is a block diagonal matrix.

The CG capacitance extraction algorithm with the $P_{intracube}$ preconditioner (PCG) is as follows:

Algorithm 1: Preconditioned CG capacitance extraction algorithm

Setup Phase.
  Divide all the conductors into a total of n tiles.
  Divide the tiles into cubes, and reorder to
    make $P_{intracube}$ block diagonal.
Compute the Potential Coefficient Matrix.
  for $i = 1$ to $i = n$
    for $j = 1$ to $j = n$
      Compute P_{ij} from (4).
  Factor $P_{intracube}$.
Loop Through all the Conductors.
  for $k = 1$ to m
    if tile i is on conductor k, set $p_i = 1$.
    else $p_i = 0$.
    Use PCG to solve $Pq = p$.
    for $l = 1$ to m $C_{kl} = \sum_{k \in conductor_l} q_k$.

Preconditioned CG (PCG).
The Setup.
  $r = p$, $q = 0$.
Conjugate-Gradient Loop.
Repeat
  Solve $P_{intracube} z = r$.
  if the first iteration $\beta = 0$.
  else $\beta = z^T r / (z^T r)_{prev}$.
  $x = z + \beta x$.
  $y = Px$.

$\alpha = \frac{z^T r}{x^T A x}$.
$q = q + \alpha x$.
$r = r - \alpha y$.
Until Converged

4 Acceleration with a Multipole Algorithm

As can be seen from examining the computation in Algorithm 1, m problems must be solved iteratively, and the major cost is computing the matrix P, and in each iteration forming the product Px, both of which are order n^2. This implies that computing the capacitance matrix with Algorithm 1 is order mn^2, and may not be much more efficient than direct factorization if the ratio of tiles to conductors is low.

An approach for reducing the cost of forming P and computing Px in the CG algorithm can be derived by recalling that if x is thought of as a charge distribution, Px is the potential due to that charge distribution. To see how this helps simplify the computation Px, consider two widely separated cubes, each with k tiles. Computing the contributions to the potentials at the center of each of the tiles in the first cube due to the k tile charges in the second cube from (4) requires k^2 calculations. If all the charges in the second cube are positive, then the k potential contributions to the first cube can be computed approximately in k operations. This is done by assuming the charges in the second cube contribute to potential in the first cube like a point charge equal to the sum of the charges in the second cube located at a "center of mass". Note that the accuracy of the approximation improves as the separation between cubes increases.

There are a collection of algorithms based on the above idea, often referred to as multipole algorithms[rohklin86, katzenelson88, zhao87]. The details of the multipole algorithm we used are well described in[greengard87], and only a very basic outline will be given here. In general, the potential, ψ, due to a cube of point charges at a location outside the radius of the cube is given by the *multipole* expansion,

$$\psi(r,\theta,\phi) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} Y_n^m(\theta,\phi) \tag{5}$$

where r, θ and ϕ are the spherical coordinates of the evaluation location, $Y_n^m(\theta,\phi)$ is the spherical harmonic, and M_n^m is the multipole coefficient.

which can be computed from the charge in the cube from

$$M_n^m = \sum_{i=1}^{k} q_i \rho_i^n Y_n^{-m}(\alpha_i \beta_i) \tag{6}$$

where ρ_i, α_i, and β_i are the spherical coordinates of the i^{th} charge. If the evaluation location is well outside the cube, then the potential can be accurately computed using just a few terms of the multipole expansion.

Consider a collection of cubes containing charges and one cube, well separated from the others, containing several locations at which the potential must be evaluated. It is possible to combine all the multipole expansions for the cubes containing charges into a single local expansion from which the potential at the evaluation points in the cube can be computed quickly. The local expansion is given by

$$v(r, \theta, \phi) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} L_n^m Y_n^m(\theta, \phi) r^n \tag{7}$$

where r, θ and ϕ are the spherical coordinates of the evaluation location, and L_n^m are the local expansion coefficients, which are computed from the combination of multipole expansions for the cubes containing charges. Good accuracy can be achieved with a few terms of the local expansion.

Truncated multipole and local expansions can be used to compute n potentials at n evaluation points in order n operations, provided the charges and evaluation points are reasonably separated. To ensure adequate separation and avoid excess calculation, careful hierarchical shifting and combining of both the multipole and local expansions is necessary, as is well described in [greengard87]. With the computation organized in this manner, the multipole algorithm can be used to compute most of Px in Algorithm 1, except the part due to interactions between tiles in a given cube, and the tiles of each cube's nearest neighbors. This implies that in Algorithm 1, if the multipole algorithm is used to compute Px, most of P need not be formed explicitly. Note also that the part that must be computed explicitly includes $P_{intracube}$, therefore the multipole accelerated PCG algorithm can still use $P_{intracube}$ as a preconditioner. Finally, note that using the multipole algorithm to compute Px implies that both n^2 steps of Algorithm 1, forming all of P and computing Px, can be removed.

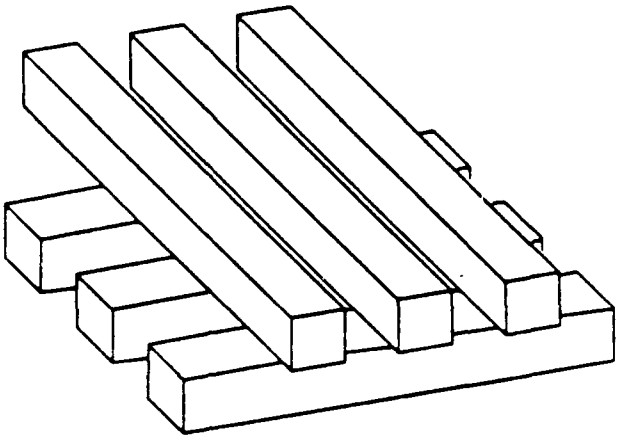

Figure 1: Bus Structure Example with Six Conductors

5 Results and Conclusions

The multipole accelerated PCG algorithm was implemented and tested on a simple bus structure (Figure 1), with 2, 4, and 6 conductors. In Table 1 we report the total number of tiles resulting from the conductor surface discretization, the number of cubes into which space was divided, the time to compute capacitance using direct factorization, PCG, and multipole accelerated PCG (MPCG), the number of iterations to achieve convergence with PCG and MPCG, and the relative error introduced by the multipole approximation.

Much additional work is under way to improve the efficiency of our MPCG-based capacitance extraction program, and CPU time comparisons for an efficient implementation will be presented at the conference. Future research includes extending the approach to piecewise-constant dielectrics and problems with ground planes.

The authors would like to thank David Ling and Albert Ruehli of the I.B.M. T. J. Watson Research Center for the many discussions that led to the approach presented here, as well as their help along the way. In addition we would like to acknowledge the helpful discussions with Jacob Katzenelson, and finally we thank the many members of the MIT Custom Integrated Circuits group for their help and encouragement.

This work was supported by the Defense Advanced Research Projects

	2 Cond.	4 Cond.	6 Cond.
tiles	216	720	1512
cubes	64	64	64
direct time	16.7	258	1810
PCG time	15.7	218	1055
MPCG time	15	123	490
PCG iters	5	7	8
MPCG iters	5	9	10
MPCG rel. err.	0.002	0.001	0.002

Table 1: Comparison of Extraction Methods

Agency contract N00014-87-K-825, and grants from IBM and Analog Devices.

References

[golub83] G. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, Maryland, 1983.

[greengard87] L. Greengard, V. Rohklin, "A Fast Algorithm for Particle Simulations," J. Comp. Phys., Vol 73, pp. 325–348, 1987.

[katzenelson88] J. Katzenelson, *Computational Structure of the N-body Problem*, Mass. Inst. of Tech., Artificial Intelligence Laboratory, AI Memo 1042, April 1988.

[ning88] Z.-Q. Ning and P. M. Dewilde, "SPIDER: Capacitance Modeling for VLSI Interconnections," *IEEE Transactions on Computer-Aided Design*, vol. CAD-7, No. 12, December 1988.

[rao84] S. Rao, T. Sarkar, R. Harrington, "The Electrostatic Field of Conducting Bodies in Multiple Dielectric Media," *IEEE Transactions on Microwave Theory and Techniques*, vol. MTT-32, No. 11, November 1984.

[rohklin86] V. Rohklin, "Rapid Solution of Integral Equation of Classical Potential Theory," J. Comput. Phys., Vol. 60, pp. 187–207, 1985.

[ruehli73] A. Ruehli and P. A. Brennen, "Efficient capacitance calculations for three-dimensional multiconductor systems," *IEEE Transactions on Microwave Theory and Techniques*, vol. MTT-21, No. 2, pp. 76–82, February 1973.

[zhao87] F. Zhao, *An O(N) algorithm for three-dimensional N-body simulations*, Master's thesis, Mass. Inst. of Tech., Dept. of Elec. Eng. and Comp. Sci., October 1987.